Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Erik André Jakobsen

Obtaining Numerical Solutions to Ordinary Differential Equations on Differentiable Manifolds with Lie Group Integrators

Master's thesis in Applied Physics and Mathematics
Supervisor: Brynjulf Owren
June 2021

NTNU
Norwegian University of Science and Technology

Erik André Jakobsen

Obtaining Numerical Solutions to Ordinary Differential Equations on Differentiable Manifolds with Lie Group Integrators

Master's thesis in Applied Physics and Mathematics
Supervisor: Brynjulf Owren
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

NTNU
Norwegian University of
Science and Technology

Abstract

Ordinary differential equations (ODEs) are at the core of a number of subjects, including dynamics, finance, and predicting the weather. The familiar Runge-Kutta methods are one of the most common ways of obtaining numerical solutions to ODEs. However, the numerical solutions provided by these methods may exhibit qualities that do not align with what one would expect through analysis of the system of equations. One example is when the exact solution is known to belong to some submanifold $\mathcal{M}$ of the ambient space in which the problem is represented.

The goal of this thesis is to present PyLie, a Python framework for a class of numerical integrators known as the Runge-Kutta Munthe-Kaas methods. Based upon Lie groups and their associated Lie algebras, these methods guarantee the conservation of invariants that may not be satisfied by the standard Runge-Kutta methods. The basic theory of such integrators is introduced, followed by the approach chosen for implementing them in Python.

The framework is applied to three different problems from dynamics and compared to the standard Runge-Kutta method as implemented in the standard Python package SciPy. It is seen that PyLie performs as expected with respect to the order of the methods and the conservation of the equation's invariants, outperforming SciPy.

The thesis concludes with suggestions for further improvements that may be made to the software.

Samandrag

Ordinære differensiallikningar (ODE-ar) er sentralt i fleire fagfelt; til dels mekanikk, finans, og det å lage vêrmeldingar. Dei velkjende Runge-Kutta metodane er ein av dei vanlegaste måtane å finne numeriske løysingar til ODE-ar på. Det kan likevel hende at desse løysingane har eigenskapar som ikkje svarar til det ein forventar ved å analysere likningane. Eit døme er når det er kjend at den analytiske løysinga høyrer til eit undermangfald $\mathcal{M}$ av det omkringliggjande rommet problemet er representert i.

Målet i denne avhandlinga er å introdusere PyLie, eit Python-rammeverk for å løyse ODE-ar ved bruk av ein type numeriske integratorar, kjend som Runge-Kutta Munthe-Kaas-metodane. Desse metodane er basert på Lie-grupper og deira tilhøyrande Lie-algebraar, og garanterer å konservere invariantar som ikkje nødvendigvis blir tilfredsstilt av dei vanlege Runge-Kutta-metodane. I avhandlinga blir den grunnleggjande teorien for metodane introdusert, etterfølgd av ei skildring av korleis dei vart implementert i Python.

PyLie brukast til å løyse tre ulike problem frå mekanikk, og samanliknast med implementeringa av ein Runge-Kutta-metode i det standard Python-biblioteket SciPy. Ein ser at PyLie gjer det som forventa når det gjeld orden for metodane, og å konservere på invariantar. Sistnemnde er i motsetnad til SciPy.

Avhandlinga konkluderer med forslag til framtidige forbetringar av programvara.

Preface

This thesis is the culmination of my seven years at the Norwegian University of Science and Technology, of which have I have spent five as a student at the programme for Applied Physics and Mathematics.

I would like to thank my supervisor, Brynulf Owren, for his continued encouragement and support during the work with this thesis.

I am indebted to my family who have always aided me and cheered me on in all of my pursuits.

Finally, I am grateful to the lovely people who have made these past years so enjoyable—most of all Karen.

La ikke byen få ro.

Contents

List of Figures

Table of Notation

The following notation is used in the thesis, tabulated here for convenience.

Notation	Meaning
$\mathcal{M}$	A manifold.
$\mathfrak{X}(\mathcal{M})$	The set of all vector fields over the manifold $\mathcal{M}$.
$T_m\mathcal{M}$	The tangent space of $\mathcal{M}$ at the element m.
$T\mathcal{M}$	The tangent bundle of $\mathcal{M}$.
G	A Lie group.
$\mathfrak{g}$	The Lie algebra associated to the Lie group G.
$\Lambda(g, m)$	The action of the Lie group element g on the manifold element m.
exp	The exponential map from a Lie algebra $\mathfrak{g}$ to its associated Lie group G.
$\lambda(x, m)$	The algebra action given by $\Lambda(\exp(x), m)$.
$\dot{y}(t)$	The derivative of y with respect to time t.
y_n	An approximation to the function $y\colon \mathbb{R} \to \mathbb{R}^n$ evaluated at time t_n.
S^n	The hypersphere of radius 1 embedded in $\mathbb{R}^{n+1}$.
$g(n) = \mathrm{O}(f(n))$	Asymptotic notation, i.e. $\lvert g(n)\rvert \leq a f(n)$ for some $a > 0$ for all $n > n_0$ for some n_0.

Introduction

The goal of this master's thesis is two-fold: The first is to serve as an introduction to numerically solving ordinary differential equations on non-linear manifolds with the use of Runge-Kutta Munthe-Kaas methods. The second is to describe the process of implementing these methods in a software package intended for anyone interested in the subject matter and to demonstrate its use on a number of examples.

Chapter 1 describes the issue of differential equations on manifolds, introduces the notation used, and motivates the need for specialized geometric integrators.

Chapter 2 introduces Runge-Kutta Munthe-Kaas integrators as a way to adapt existing Runge-Kutta methods to non-linear manifolds using Lie groups and Lie algebras.

Chapter 3 discusses the implementation of these methods in Python using an object-oriented approach. The source code is freely available, and it is described how to extend it to solve problems that may not yet be supported by the software.

Chapter 4 demonstrates the use of the software on three problems of increasing complexity and compares the results to a standard integrator from the Python library SciPy.

Finally, Chapter 5 concludes the thesis with a summary of the work done so far and suggestions for further improvements that still remain as future work.

Chapter 1

Ordinary Differential Equations on Manifolds

This chapter mostly contains basic theory on ordinary differential equations (ODEs) and the use of Runge-Kutta methods to solve them. We expect that much of the material will be familiar to the reader, but include it here to define the notation used and to motivate the need for geometrical numerical integrators.

This chapter presents some algorithms in the form of pseudocode. The first line presents the name of the algorithm typeset in a SMALL CAPS font, followed by a list of the input arguments. The following numbered lines describe the behaviour of the algorithm. Algorithms will be referred to by their name.

1.1 Numerical solutions of ODEs in general

A general ordinary differential equation is a system of equations given by

$$\begin{aligned} \dot{y}(t) &= f(t, y(t)), \\ t &\in [0, T], \\ y(0) &= y_0 \end{aligned} \tag{1.1}$$

for some $y : \mathbb{R} \to \mathbb{R}^n$, $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$, $T > 0$, $n \geq 1$. We use the notational convention

$$\dot{y}(t) \coloneqq \frac{\mathrm{d}y}{\mathrm{d}t}(t),$$

which in the case where $y \in \mathbb{R}^n$ for $n > 1$ is equivalent to

$$\dot{y}(t) = \frac{\mathrm{d}y}{\mathrm{d}t}(t) = \left[\frac{\mathrm{d}y_1}{\mathrm{d}t}(t), \quad \frac{\mathrm{d}y_2}{\mathrm{d}t}(t), \quad \dots \quad \frac{\mathrm{d}y_n}{\mathrm{d}t}(t) \right]^T.$$

In the general case of (1.1), several approaches exist for calculating approximate numerical solutions. In the following, y_n denotes the numerical approximation

to the exact solution at time t_n, i.e. $y_n \approx y(t_n)$. A large number of methods take the approach of assuming $y(t_n)$ is known for some $t_n \in [0, T]$, and calculating the "direction" of the solution y at time $t = t_n$ by some approximation technique. The numerical approximation to the solution is obtained by advancing the value of $y(t_n)$ by a step length h in this direction. When the problem is an initial value problem, as in (1.1), the value of the solution will always be known at $t = 0$. This naturally extends to iterative numerical solvers: Starting from $t_0 = 0$, obtain the approximate solution y_1 at $t_1 = h > 0$ using the procedure above. Then repeat to find $y_2 \approx y(2h), y_3 \approx y(3h), \ldots$ until reaching $y_N \approx y(T)$. We refer to each iteration as taking a *step*, and the scalar parameter $h > 0$ is referred to as the *step length*. A numerical method of this kind is thus completely defined by describing the computations involved in taking a step of length h.[1] All algorithms presented in this thesis will adhere to this convention and only present the computation involved in a single step, eliminating the need for boilerplate code in the pseudocode.

The simplest method using the general framework presented above is the Explicit Euler method, in which a single step takes the following form:

ExplicitEuler(f, t_n, y_n, h)
1 $k_1 = f(t_n, y_n)$
2 **return** $y_n + hk_1$

The seemingly arbitrary choice of assigning the value of $f(t_n, y_n)$ to the variable k_1 will become clear as we move on to similar methods of greater complexity. Each step of the explicit Euler method considers the direction of the solution trajectory at the given point y_n, and moves the solution in a step of length h in this direction as illustrated in Figure 1.1. The term *explicit* in the name of the method refers to the fact that we are only using information about the solution that is available at the current time. This is in contrast to *implicit* methods, which involve solving an (in general non-linear) equation at each time-step [17]. For instance, the *implicit Euler's method* is given by

ImplicitEuler(f, t_n, y_n, h)
1 solve $k_1 = f(t_{n+1}, y_{n+1})$ with respect to k_1
2 **return** $y_n + hk_1$

In line one, the unknown value of y_{n+1} must be found by solving an equation at each step. As the work in this thesis will only concern explicit methods, we will sometimes use simply *Euler's method* to refer to the explicit Euler's method.

A slightly more involved explicit algorithm is *Heun's method*, also known as the *explicit trapezoidal rule*:

[1] This does not include the choice of h, which in some methods is also adaptively updated in each step by the numerical method in order to balance the numerical error with the computational cost.

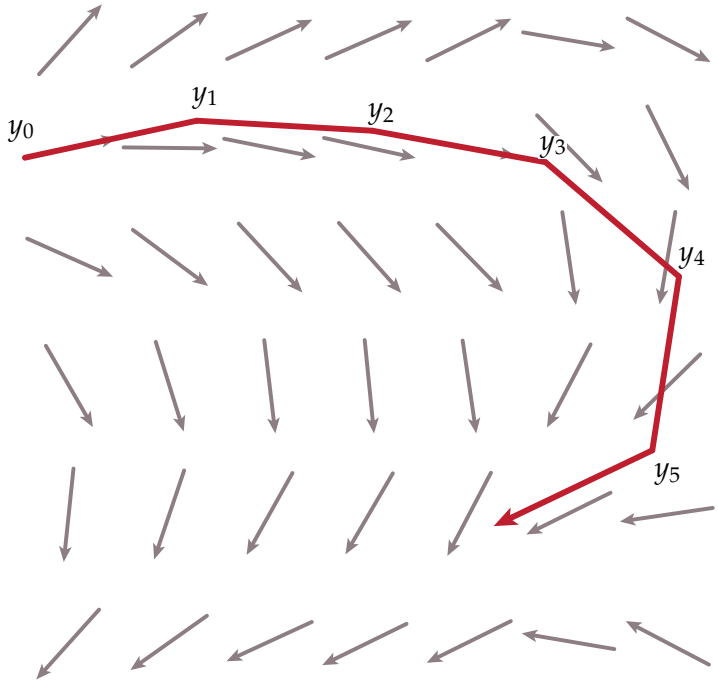


Figure 1.1: Illustration of the explicit Euler method in a vector field in $\mathbb{R}^2$. The red line is the numerical solution, where each straight segment (y_n, y_{n+1}) is of length $\|y_{n+1} - y_n\| = h$.

HEUNS(f, t_n, y_n, h)
1 $k_1 = f(t_n, y_n)$
2 $k_2 = f(t_n + h, y_n + hk_1)$
3 **return** $y_n + \frac{h}{2}(k_1 + k_2)$

Here, as in Euler's method, we begin by considering the trajectory of the solution at the current point. One additional step is then performed by using this value to produce an estimate of $f(t_{n+1}, y_{n+1})$. Intuitively, this may be seen as averaging the trajectory at two points to produce what is hopefully a better estimate of y_{n+1}.

1.2 Numerical convergence

One desirable property of the numerical approximations is that they are, in some sense, *close* to the exact solution of the ODE in question. Given an iterative method, define y_N^h as the numerical solution of (1.1) at time t_N, taking however many steps N necessary to reach t_N with step length h. The *global*

error of a method is defined as

$$e_{N,h} = y_N^h - y(t_N),$$

where $y(t_N)$ is the exact solution at time t_N. For ease of notation we will often drop the subindex N, and simply write e_h. A method is said to be convergent if

$$\lim_{h \to 0} \|e_h\| = 0,$$

see [37]. It may be shown that all methods considered in this thesis has a global error which for smooth problems can be written as

$$e_h = e_p h^p + e_{p+1} h^{p+1} + \ldots = \mathrm{O}(h^p) \tag{1.2}$$

where $p \geq 1$ is an integer and e_p, e_{p+1} are real coefficients. The number p is called the *order* of the method, and is the primary quantity of interest when comparing iterative numerical methods.

1.3 Runge-Kutta Methods

The ideas behind ExplicitEuler and Heuns can be generalized to produce estimates of arbitrary accuracy, collectively known as the *Runge-Kutta methods* [35]. The algorithm for a method consisting of s *stages* is as follows:

GeneralRungeKutta(f, t_n, y_n, h)
1 **for** $i = 1$ **to** s
2 solve $k_i = hf\left(t_n + c_i h,\ y_n + \sum_{j=1}^{s} a_{ij} k_j\right)$ with respect to k_i
3 **return** $y_n + \sum_{i=1}^{s} b_i k_i$

The constants b_i, c_i and a_{ij} are all parameters of the particular method in question. Note that if $a_{ij} = 0$ for all $j \geq i$ the method is explicit. The algorithm may then be simplified to

ExplicitRungeKutta(f, t_n, y_n, h)
1 **for** $i = 1$ **to** s
2 $k_i = hf\left(t_n + c_i h,\ y_n + \sum_{j=1}^{i-1} a_{ij} k_j\right)$
3 **return** $y_n + \sum_{i=1}^{s} b_i k_i$

The parameters of ExplicitRungeKutta may be efficiently presented in a *Butcher tableau*:

$$\begin{array}{c|cccc} c_1 & & & & \\ c_2 & a_{21} & & & \\ \vdots & \vdots & \ddots & & \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array},$$

or even more compactly,

$$\begin{array}{c|c} c & A \\ \hline & b^T \end{array}.$$

The condition for the method defined by a given tableau being explicit is now that the matrix A is strictly lower-triangular.

The tableaus for Euler's and Heun's method are, respectively

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array} \quad \text{and} \quad \begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \end{array}.$$

One of the most widespread Runge-Kutta methods, known simply as *Runge-Kutta-4* (RK4), is given by

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}.$$

As the name suggests, this is a method of order 4 [16].

1.4 Manifolds

So far, we have assumed that for an ODE $\dot{y} = f(t, y)$ where $y \in \mathbb{R}^n$, all points $u \in \mathbb{R}^n$ are permissible values for a solution. This is not always the case. If, for instance, $y(t)$ is the position of a mass connected to the origin by a rigid rod of length ℓ (i.e. a pendulum), we would expect that $\|y(t)\|_2 = \ell$ for all times t. Another example might be a physical system where $y(t) \in \mathbb{R}^n$ encodes the position and momentum of N particles—in this case, the conservation of energy constrains the possible values of y to a subset of $\mathbb{R}^n$.

For a practical example of the above issues, we present the equations for the mathematical pendulum in a Hamiltonian framework with momentum

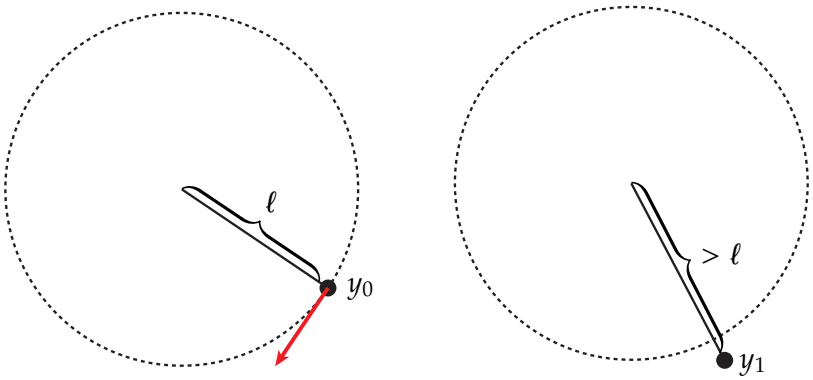


Figure 1.2: Illustration of a situation where explicit Euler fails. In (a) we see the initial state for a pendulum with a rod of length ℓ under the influence of gravity. The resulting force is orthogonal to the circle of radius ℓ. In (b) we have taken a single step with explicit Euler, which yields a numerical approximation with a rod with length greater than ℓ.

$p(t)$ and position $q(t)$. The equations of motion are given by

$$\dot{p} = -\sin q, \qquad \dot{q} = p. \tag{1.3}$$

The energy of the system, given by

$$H(p, q) = \frac{1}{2}p^2 - \cos q \tag{1.4}$$

is seen to be constant, as

$$\frac{\mathrm{d}H}{\mathrm{d}t} = \frac{\partial H}{\partial p}\frac{\mathrm{d}p}{\mathrm{d}t} + \frac{\partial H}{\partial q}\frac{\mathrm{d}q}{\mathrm{d}t} = p\dot{p} + \sin(q)\dot{q} = \dot{q}\cdot\dot{p} - \dot{p}\cdot\dot{q} = 0.$$

In other words, the curve $y(t) = \big(p(t), q(t)\big)$ coincides with the level curves of H.

However, the numerical integrators described in the previous section provide no guarantee that these constraints are respected. Indeed, Figure 1.3 shows the result of solving (1.3) with ExplicitEuler. The obtained solution immediately diverges, leading to an unsatisfactory result of little practical or theoretical use. A simple illustration of the issue in this particular example is given by Figure 1.2.

To formalize the issue, we denote by $\mathcal{M} \subset \mathbb{R}^n$, $d \leq n$, a d-dimensional differential manifold—that is, a d-dimensional topological space equipped with continuous local coordinate charts $\phi_i \colon U_i \subset \mathcal{M} \to \mathbb{R}^d$ such that all

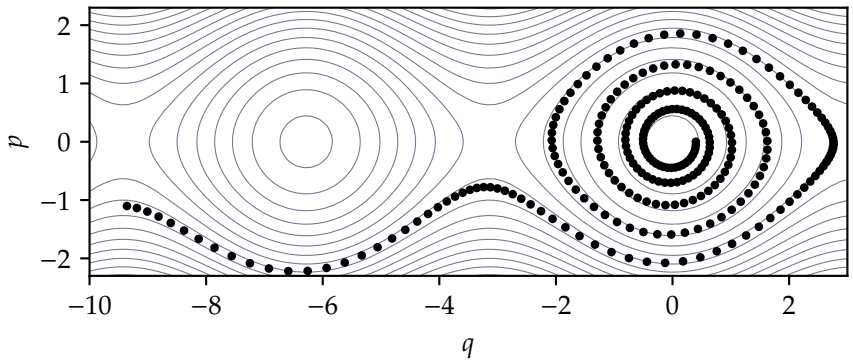


Figure 1.3: Numerical solution (black dots) of the mathematical pendulum (1.3), obtained with Euler's method $y_{n+1} = y_n + hf(t_n, y_n)$. The initial condition was set to $p_0 = 0$, $q_0 = 0.4$, the step length to $h = 0.15$, and the method took 260 steps. The solid lines are the level curves of (1.4), which coincide with the analytical solution for various initial values of the problem. The numerical solution clearly diverges.

overlapping charts $\phi_{ij} \colon \mathbb{R}^d \to \mathbb{R}^d$ defined by $\phi_j \circ \phi_i^{-1}$ restricted to $\phi_i(U_i \cap U_j)$ are diffeomorphisms [1]. For our purposes, however, it is sufficient to think of manifolds as topological spaces where for all $m \in \mathcal{M}$ there is a neighbourhood U containing m which "looks like" Euclidean space.

One keyword in the above definition is *differentiable*. This means that for all $m \in M$, there is a vector space $T_m\mathcal{M}$ called the *tangent space of* $\mathcal{M}$ *at* m [25]. One useful way to define this is through curves: Let $\gamma(t)$ be a smooth curve in $\mathcal{M}$ such that $\gamma(0) = m$. We may then take the time-derivative of γ at $t = 0$ to obtain a velocity vector v_m, i.e. $v_m = \dot{\gamma}(0)$; see Figure 1.4. The tangent space of a manifold $\mathcal{M}$ at the point m is a vector space given by the velocities of all such curves at the point m. The following definition is taken from [15]:

$$T_m\mathcal{M} := \left\{ v \in \mathbb{R}^n \;\middle|\; \begin{array}{l} \text{there exists a continuously differentiable } \gamma \colon (-\varepsilon, \varepsilon) \to \mathbb{R}^n \text{ such that } \gamma(t) \in \\ \mathcal{M} \text{ for } t \in (-\varepsilon, \varepsilon) \text{ and } \gamma(0) = m \text{ and} \\ \dot{\gamma}(0) = v \end{array} \right\}.$$

The *tangent bundle* of $\mathcal{M}$ is given by the (disjoint) union of the tangent spaces of each point of the manifold:

$$T\mathcal{M} = \bigcup_{m \in \mathcal{M}} T_m\mathcal{M}.$$

In general the tangent bundle $T\mathcal{M}$ is not a linear space.

A more detailed description of manifolds may be found in e.g. [6, 25, 38].

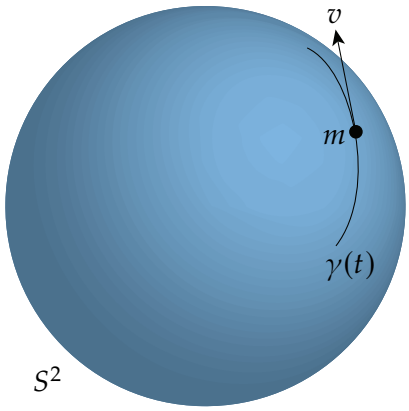


Figure 1.4: A curve $\gamma(t) \in S^2$ with $\gamma(0) = m$ generating a vector $\dot{\gamma}(0) = v \in T_m S^2$.

Examples of differentiable manifolds include the n-sphere S^n for $n \geq 2$, the real line $\mathbb{R}$, and Euclidean space $\mathbb{R}^d$ itself.

We may now define a differential equation on a manifold $\mathcal{M} \subset \mathbb{R}^n$ using the tangent bundle $T\mathcal{M}$. A vector field on $\mathcal{M}$ is a map $f : \mathbb{R} \times \mathcal{M}$ (alternatively, $f : \mathcal{M} \to \mathbb{R}^n$) such that

$$f(t, y) \in T_y\mathcal{M} \qquad \text{for all } y \in \mathcal{M}.$$

A vector field on $\mathcal{M}$ is called a *section* of $T\mathcal{M}$, and we write $f \in \mathfrak{X}(\mathcal{M})$ [34]. Given such a vector field,

$$\dot{y}(t) = f(t, y) \tag{1.5}$$

is a differential equation on $\mathcal{M}$. A function $y : \mathbb{R} \to \mathcal{M}$ that satisfies (1.5) is called an *integral curve* or simply the solution to the equation [14].

We have already seen that applying any numerical method and simply hoping for the good fortune of a solution in $\mathcal{M}$ will not do. The goal is thus to leverage the knowledge of $\mathcal{M}$ and $f(t, y)$ to produce a numerical method approximation to (1.5) that is guaranteed to satisfy $y \in \mathcal{M}$. This is the topic of the next chapter.

Chapter 2

Geometric numerical integration

The goal of geometric numerical integration is to preserve the geometric properties of the problem. As an example, consider an ordinary differential equation on the form

$$\dot{y}(t) = A(t, y) \cdot y, \qquad y(0) = y_0 \tag{2.1}$$

where $y \in \mathbb{R}^n$ and $A \colon \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^{n \times n}$ maps (t, y) to a skew-symmetric matrix. One property of the solution to this equation is that $\|y(t)\|_2 = \|y_0\|_2$ for all t.[1]

At $t = 0$ we may approximate (2.1) by "freezing" the vector field:

$$\dot{y}(t) = A(0, y_0)y.$$

The exact solution is

$$y(t) = \exp\big(tA(0, y_0)\big)y_0 \tag{2.2}$$

where $\exp\big(tA(0, y_0)\big)$ is the *matrix exponential*, defined as

$$\exp A = \sum_{k=0}^{\infty} \frac{A^k}{k!} = I + A + \frac{1}{2!}A^2 + \dots. \tag{2.3}$$

The matrix exponential is sometimes also denoted by e^A.

Equation (2.2) defines the simplest iterative Lie group integrator, known as the Lie-Euler method:

LieEuler(f, t_n, y_n, h)
1 **return** $\exp(hf(t_n, y_n)) \cdot y_n$

It can be shown that when A is a skew-symmetric matrix, $\exp A$ will be orthogonal [19]. Thus, LieEuler ensures that $\|y_{n+1}\|_2 = \|y_n\|_2$.

For now, we are restricted to equations of the type (2.1). The goal of this chapter is to generalize the expression to differential equations described by elements of a Lie group acting on the configuration manifold of y.

[1] Note that $\frac{\mathrm{d}}{\mathrm{d}t}\frac{1}{2}\|y(t)\|_2^2 = y^T A y = -y^T A y$, implying $\frac{\mathrm{d}}{\mathrm{d}t}\frac{1}{2}\|y(t)\|_2 = 0$ and $\|y(t)\| = \textit{constant}$.

2.1 Lie groups and Lie algebras

A *Lie group* $(G, \cdot)$ is a differential manifold that also has the structure of a group where the maps $(p, q) \mapsto p \cdot q$ and $p \mapsto p^{-1}$ are smooth for all $p, q \in (G, \cdot)$ [22]. We denote the identity element of a Lie group by e. A large class of Lie groups is the matrix Lie groups, where the group product is the usual matrix multiplication. Some examples include:

GL(n) The general linear group of all invertible $n \times n$ matrices.

SL(n) The special linear group of all invertible $n \times n$ matrices with determinant one.

SO(n) The special orthogonal group of all orthogonal $n \times n$ matrices with determinant one.

In this thesis, we simplify the notation and use G to refer to a general Lie group.

A *Lie algebra* $\mathfrak{g}$ is a vector space eqiupped with a bilinear skew-symmetric map $[\,\cdot\,,\,\cdot\,]\colon \mathfrak{g} \times \mathfrak{g} \to \mathfrak{g}$ that satisfies the Jacobi identity:

$$[x,[y,z]] + [y,[z,x]] + [z,[x,y]] = 0 \quad \text{for all } x, y, z \in \mathfrak{g}.$$

This map is called the *Lie bracket* [10]. For convience, we define $\mathrm{ad}_x\colon \mathfrak{g} \to \mathfrak{g}$ to be the linear map $\mathrm{ad}_x(y) = [x, y]$ which allows for repeated applications of the Lie bracket:

$$\begin{aligned} &\mathrm{ad}_x^0(y) = y, \\ &\mathrm{ad}_x^n(y) = \mathrm{ad}_x(\mathrm{ad}_x^{n-1}(y)) = \underbrace{[x,[x,x,[\ldots[x}_{n \text{ times}},y]]]], \quad n \geq 1. \end{aligned}$$

Each Lie group has an associated Lie algebra defined as the tangent space of the group at the identity element; that is, the Lie algebra $\mathfrak{g}$ associated with the Lie group G is defined by $\mathfrak{g} = T_e G$. The Lie bracket is then given by

$$[x, y] = \left.\frac{\partial^2}{\partial t \partial s}\right|_{t=s=0} g(t)h(s)g^{-1}(t)$$

where $g(t)$ and $h(s)$ are curves in G such that $g(0) = h(0) = e$, $g'(0) = x$, $h'(0) = y$. If G is a matrix Lie group, the Lie bracket is the matrix commutator

$$[x, y] = x \cdot y - y \cdot x.$$

The (left) Lie group action of a Lie group G on a manifold $\mathcal{M}$ is a smooth map $\Lambda\colon G \times \mathcal{M} \to \mathcal{M}$ which satisfies

$$\begin{aligned} \Lambda(e, m) &= m \text{ for all } m \in \mathcal{M}, \text{ and} \\ \Lambda(g \cdot h, m) &= \Lambda(g, \Lambda(h, m)) \text{ for all } g, h \in G \text{ and } m \in \mathcal{M}. \end{aligned}$$

Given a element $m \in \mathcal{M}$ the *orbit* of m, $\mathcal{O}(m) \subseteq \mathcal{M}$ is given by

$$\mathcal{O}(m) = \left\{ \Lambda(g, m) \colon g \in G \right\},$$

see for instance [32]. If $\mathcal{O}(m) = \mathcal{M}$, the group action is said to be *transitive*. In particular, this means that for any pairs $m_1, m_2 \in \mathcal{M}$ there is a $g \in G$ such that $\Lambda(g, m_1) = m_2$. A manifold $\mathcal{M}$ equipped with a transitive Lie group action is said to be *homogenous* [42].

Note that any $\xi \in \mathfrak{g}$ specifies a tangent $\xi_m \in T_m\mathcal{M}$ by

$$\xi_m \coloneqq \lambda_*|_m(\xi) = \left.\frac{\mathrm{d}}{\mathrm{d}t}\right|_{t=0} \Lambda(\exp(t\xi), m), \tag{2.4}$$

where $\exp \colon \mathfrak{g} \to G$ is defined as $\exp(\xi) = \sigma(1)$, where σ is in turn the solution to the ODE

$$\dot{\sigma}(t) = \xi\sigma(t), \qquad \sigma(0) = e.$$

In the case of G being a matrix Lie group, exp is the matrix exponential defined in (2.3). The operator λ_* in (2.4) is called the *infinitesimal generator* of the action.

Define the *algebra action* $\lambda \colon \mathfrak{g} \times \mathcal{M} \to \mathcal{M}$ by

$$\lambda(\xi, m) = \Lambda(\exp(\xi), m).$$

If the algebra action is transitive, any ODE on $\mathcal{M}$ may be written as

$$\dot{y}(t) = \lambda_*|_y(f(t, y)), \qquad y(0) = y_0 \tag{2.5}$$

for some $f \colon \mathbb{R} \times \mathcal{M} \to \mathfrak{g}$; moreover, finding a (locally) transitive algebra action will always be possible [28]. For sufficiently small t, the solution to (2.5) is given by

$$y(t) = \lambda(u(t), y_0) \tag{2.6}$$

where $u(t) \in \mathfrak{g}$ satisfies

$$\dot{u}(t) = \operatorname{dexp}_u^{-1}\left(f(t, \lambda(u, y_0))\right), \qquad u(0) = 0. \tag{2.7}$$

Here, $\operatorname{dexp}_u^{-1}$ is the map

$$\operatorname{dexp}_u^{-1} = \left.\frac{z}{e^z - 1}\right|_{z=\operatorname{ad}_u} = \sum_{n=0}^{\infty} \frac{B_n}{n!} \operatorname{ad}_u^n \tag{2.8}$$

where B_n is the nth Bernoulli number.

The idea behind the Lie group integrators is now as follows, assuming y_n is available:

1. Formulate the ODE in the form of (2.5).

2. Use a standard RK method to find a numerical solution u_{n+1} to (2.7). The value of $\operatorname{dexp}_u^{-1}$ may either be calculated using an exact formula, or, if this is not available, calculated using a truncated series with $p-1$ terms for a RK method of order p [10]. If $p \leq 2$, this means $\operatorname{dexp}_u^{-1}$ may be replaced with the identity map.

3. Advance the numerical solution on $\mathcal{M}$ by $y_{n+1} = \lambda(u_{n+1}, y_n)$.

For the first step, y_0 is given as the initial condition. This translates to the following algorithm, with the constants a_{ij}, b_i and c_i being the coefficients of any standard explicit Runge-Kutta method:

RungeKuttaMuntheKaas(f, t_n, y_n, h)
1 **for** $i = 1, 2, \ldots, s$
2 $\quad u_i = h \sum_{j=1}^{i-1} a_{ij} k_j$
3 $\quad k_i = \operatorname{dexp}_{u_i}^{-1} f\left(t_n + c_i h, \lambda(u_i, y_n)\right)$
4 $v = h \sum_{i=1}^{s} b_i k_i$
5 **return** $\lambda(v, y_n)$

This is referred to as a Runge-Kutta Munthe-Kaas method, abbreviated to RKMK. This class of integrators was developed by Munthe-Kaas in a series of papers in the 1990s [28, 29, 30, 31].

Table 2.1: The first few Bernoulli numbers B_n, used to define $\operatorname{dexp}_u^{-1}$ in (2.8). Note that for $n = 3, 5, 7, \ldots$, $B_n = 0$. More terms can be found at [40].

k	B_k
0	1
1	$-\frac{1}{2}$
2	$\frac{1}{6}$
4	$-\frac{1}{30}$
6	$\frac{1}{42}$
8	$-\frac{1}{30}$
10	$\frac{5}{66}$

Chapter 3

Geometric numerical integration in Python

The methods discussed in the previous chapter have been implemented in Python using object-oriented programming. This chapter will first introduce the main concepts of this style of programming. Hopefully, this will serve as sufficient motivation for why this programming paradigm is suitable for implementing numerical algorithms in general, and methods based upon Lie groups and algebras in particular. The rest of the chapter will be dedicated to the architecture and use of the PyLie software.

3.1 Object-Oriented Programming

This introduction will be kept brief and only cover the core concepts needed to understand the structure of programs written in an object-oriented style. The interested reader may find a more comprehensive introduction to object-oriented programming with examples written in Python in [11]. Readers already familiar with object-oriented programming may safely skip this section.

Basics

Object-oriented programming (OOP) is a programming paradigm where the central concept is that of an *object*. In this context, an object is a collection of data, *attributes*, and behaviours, *methods*. In several object-oriented programming languages, including Python, objects are defined as instances of *classes*. The class defines a *type* of object and the associated attributes and methods of all such objects. A particular instance then defines the value of each attribute. Both attributes and methods are accessed with dot notation: The attribute `value` on the class instance `object` is accessed with `object.value`. Similarly, methods are accessed by `object.method()` with any arguments to the method

separated by comments between the parentheses. A method may have zero or more arguments. In this text, methods will always be indicated by a pair of parentheses after the method name.

As an example, the PyLie package implements a class called `Solver`, with attributes for the matrix a and the vectors b and c associated with the Butcher tableau of a given RKMK method. One instance of this class might be `explicit_euler` where `explicit_euler.a` is the zero matrix, `explicit_euler.c` the zero vector, and `explicit_euler.b` the one-dimensional unit vector.

Note that the attributes of an object may be of any type: They may be simple types such as integers or arrays, or more complex types such as functions or other objects. The latter is used extensively in PyLie.

Inheritance and Polymorphism

A key concept of OOP is that of inheritance. In short, this means that one class of objects extends the functionality and/or the attributes of another class. The inheriting class is know as a *child class* or a *subclass* of the predefined *parent class* or *superclass*. As an example, consider a class `so3` which is a subclass of `LieAlgebra`. Any attribute or method of `LieAlgebra` will also be available to `so3`. In addition, we may wish to implement e.g. the hat map $\widehat{\cdot}$ to transform elements of the lie algebra from their $\mathbb{R}^3$ representation to a skew-symmetric 3-by-3 matrix—more on this in Section 4.1. We may then define this method in the `so3` class without affecting the `LieAlgebra` class.

Finally, we discuss the idea of *polymorphism*, where we may provide a common programming interface to perform different tasks. A simple example is the Python built-in `len()` function, which has two distinct behaviours depending on whether the input is a list or a string.

For a more sophisticated example we may return to `so3` and `LieAlgebra`. The parent class `LieAlgebra` implements a method `dexpinv(q, p, r)` method which numerically calculates an rth order approximation of $\mathrm{dexp}_q^{-1}(p)$ using the Lie bracket. However, in the case of $\mathfrak{so}(3)$, we have an analytical expression for this quantity that we would like to use instead. We may then implement a method of the same name in the class definition of `so3` to perform this calculation, which will then be used by all instances of this class.

The following inheritance rules apply:

- All attributes and methods of a parent class are available to all child classes.
- When a method of the same name is defined in both the parent class and the child class, the definition of the child class takes precedence on all instances of this class.

Both of these rules are illustrated in Figure 3.1.

A subclass may have more than one parent class. If two or more parent classes define a method of the same name, the method of the parent class specified first takes precedence.

3.2 The PyLie Package

The PyLie package is a framework for numerically solving ODEs with the techniques described in Chapter 2. It is the original work of the author and the main work of this thesis. The source code of the package is available at [23]. The package may be installed from the Python Package Index with the shell command displayed in Listing 1. The project may also be viewed at the

```
$ python -m pip install pylie
```

Listing 1: Installation of PyLie.

Python Package Index webpage [24]. PyLie is heavily inspired by the MATLAB software *DiffMan*, introduced in [10].

As previously mentioned, this section is dedicated to describing the architecture of the software and design choices made when writing the source code. The practical use of the package is demonstrated in Chapter 4. A complete code example where PyLie is used to solve an equation evolving on SO(3) is included in Appendix A.

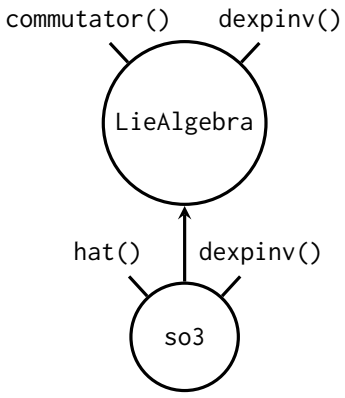


Figure 3.1: Diagram illustrating an example of class inheritance and polymorphism. The class `so3` is a child of `LieAlgebra`. The method `commutator()` is available to the child class as it is defined in the parent class. The child class definition of `dexpinv()` re-implements the method of the same name in the parent class, and is an example of polymorphism. The `hat()` method is only available on the child.

Code Structure

PyLie employs a modular design, where the central mathematical building blocks of Chapter 2 are implemented as separate objects. This includes manifolds, Lie groups, Lie algebras, and time steppers (numerical solvers).

The manifold object specifies the associated Lie group and Lie algebra. It may also enforce certain constraints that any numerical representation of elements of the manifold in question must satisfy. For instance, the implementation of $S^2 \subset \mathbb{R}^3$ will throw an error if it encounters a vector y that does not satisfy $\|y\|_2 = 1$ to within a specified numerical accuracy.

The Lie group object defines the action $\Lambda\colon G \times \mathcal{M} \to \mathcal{M}$ of the group on the manifold.

The Lie algebra object implements the exp map, the dexp^{-1} map, and in some cases, auxiliary methods such as the hat map $\hat{\cdot}$ to transform elements of the Lie algebra from one numerical representation to another. If the associated Lie group is a matrix Lie group, the Lie bracket is also available as the matrix commutator.

The time stepper object defines the parameters of the various Runge-Kutta Munthe-Kaas methods, and are perhaps the objects where inheritance proves to be the most useful. The parent class implements a method `step()` that is essentially the RungeKuttaMunthe-Kaas algorithm of the previous chapter translated to Python. One might then add any specific method with s stages to the package by creating a new class that inherits from the parent time stepper class, and defines the Butcher coefficients A as a two-dimensional $s \times s$ array, and b and c as arrays of length s.

The main purpose of this structure is flexibility. As it is implemented, the time stepper has no knowledge of how the action or the exponential map is implemented—this is delegated entirely to the Lie group and the Lie algebra. This is similar to how the Runge-Kutta Munthe-Kaas algorithm itself works: As long as the problem formulation and the underlying calculations are performed correctly, the procedure will work for any Lie group and Lie algebra.

One consideration is whether or not implementing a representation of a given manifold and the corresponding Lie group and Lie algebra as three separate classes is reasonable. Combining these three classes into one would still work fine, and would emphasize the tight coupling between these mathematical objects—especially so for the Lie group and its corresponding Lie algebra. However, separating the class definitions also has some clear advantages. If there is a need to represent a certain group G or algebra $\mathfrak{g}$ differently for a given problem, altering only the necessary parts of the code are easier with a modular design.[1] The same Lie structures could also be useful for different kinds of manifolds, which should implement varying constraints

[1] This is taken advantage of in the source code of the experiments in the following chapter, where the class defining $\mathfrak{se}(3)$ was used as a parent class to the implementation of $\mathfrak{se}(3)^N$.

```
$ python -m unittest discover src
.....
----------------------------------------------------------------
Ran 5 tests in 2.319s

OK
```

Listing 2: The command used to run the tests in PyLie and the output of the command.

to the numerical representation of their elements. Lastly, this code structure makes it clear where to find the various methods of the numerical framework when reading the source code.

Software Testing

A number of automated tests have been implemented to verify the correctness of the various methods of the PyLie package. For instance, on such test instance generates a random vector of length 1, applies the group action of a random element of SO(3) on this element, and verifies that the result of this mapping is also of length one.

The tests have been implemented with `unittest`, a part of the Python Standard Library [13]. To run the tests, navigate into the root folder of the PyLie package and run the shell command shown in Listing 2.

Solving a Differential Equation

In order to use the package to solve a differential equation, the function `solve()` is provided. It accepts the following arguments:

`f` The function defining the differential equation in the form (2.5).

`y` The initial value y_0.

`t_start` Initial time.

`t_end` End time.

`h` Step length.

`manifold` Text string corresponding to one of the supported manifolds.

`method` Text string corresponding to one of the supported timesteppers.

When this function is called, the following happens:

1. The specified manifold object is created, and with it, the correct Lie group and Lie algebra.

2. The specified time stepper is initialized with the appropriate implementations of the exp, dexp and Λ maps as defined by the Lie group and Lie algebra classes associated with the manifold.

3. The required number of steps N is calculated. If necessary, the final steplength is adjusted to be shorter.

4. The time stepper is used to calculate a numerical solution at each required step.

5. The solution is returned packaged in an object with two attributes:

   - `Y`, the solution represented as a two-dimensional array with each of the $N + 1$ columns representing the solution at one step.
   - `T`, an array of length $N + 1$ containing the times at which the solution was obtained.

The hierarcy of the process is illustrated in Figure 3.2.

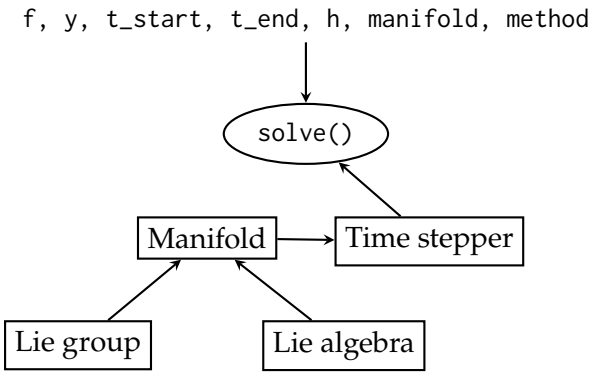


Figure 3.2: Data flow when solving an ordinary differential equation with PyLie. The function is represented with an ellipse, whilst the objects are represented with rectangles. Information flows in the directions of the arrows. In other words, the objects *Lie group* and *Lie algebra* provide their function definitions to the *Manifold* object, which in turn pass them on to the *Timestepper* object.

Implementing a New Manifold, Lie Group and Lie Algebra

In order to solve a problem on a manifold or with a Lie structure that is not yet implemented to PyLie, the following classes and corresponding methods must be implemented by the user after obtaining the source code from [23].[2]

- A manifold class, representing $\mathcal{M}$. This class must be initialized with an element $y \in \mathcal{M}$. The corresponding Lie group and Lie algebra must be specified. If desired, the class may verify that $y \in \mathcal{M}$ by checking against some specified constraints. The manifold may also have an optional `description` attribute, a short string describing the manifold. See Listing 3 for an example.

- A Lie group class. Must implement the action $\Lambda\colon G \times \mathcal{M} \to \mathcal{M}$, see Listing 4.

- An optional Lie algebra class that implements analytical expressions for the exp and/or the dexp_u^{-1} maps. If the elements of the Lie algebra are represented as matrices, the implementation of this class may be skipped and the manifold may instead use the base `LieAlgebra` class. This class uses the SciPy implementation of the matrix exponential [27, 39], as well as a truncated series of matrix commutators to calculate exp and dexp_u^{-1}, respectively.

In addition, the names of each newly implemented class must be made available to functions in other files of the package. This is done through importing them into the `__init__.py` files and adding their names as strings to the `__all__` list defined in this file. This is due to the way Python modules work and communicate with each other, see [12]. An example is shown in Listing 6.

Another requirement is that the methods are *compatible* with each other. In short, this simply means that the representations of the elements of G, $\mathfrak{g}$ and $\mathcal{M}$ are consistent across the different classes, so that a function calls combining them. e.g. `group.action(algebra.exp(g), m)`, will function without error.

Finally, the manifold must be made available to the `solve()` function. This is done by importing it into the file `solve/solve.py`, and then adding the imported object to the dictionary `_MANIFOLDS` with an appropriate string as its key. This automatically adds it to the output of the function `pylie.manifolds()`.

Implementing a New Time Stepper

If the required time stepper fits in the Runge-Kutta Munthe-Kaas as described in the previous chapter, implementing a new time stepper is straightforward. The new class must inherit from the existing 'TimeStepper' class, accept a

[2] Some short examples of implementing new objects will be displayed in the included code listings, but the best examples are found in the actual implementations in the source code.

manifold instance when initialized, and pass this on to the parent. In addition, the class must define the Butcher tableau parameters A, b and c, where A must be a strictly lower-triangular $s \times s$ array, and b and c must be one-dimensional arrays of length s. The number of stages and the order of the method must also be specified for the `solve()` method to function correctly. See Listing 7 for an example of implementing Heun's method.

Another option is to implement a custom time stepper that is not formulated with the RKMK framework, but still utilizes the structure of Lie groups. In this case, the class must also define a `step()` method that calculates y_{n+1} given f, t_n, y_n, and the step length h. The initialization method of the custom time stepper must accept a manifold object as its argument, but it does not have to use it. It cannot rely on other arguments without further modifications to the entire PyLie package. At that point, it might be better to write the solver as a standalone program. For an example of how to structure a custom time stepper, see Listing 8.

In any case, the time stepper must be made available to the `solve()` function. This is done in a similar manner as with the manifold object: Import the time stepper object and add its name as a string to the `__all__` list in the file `timestepper/__init__.py`. Finally, import it into `solve/solve.py` and add it to the `_METHODS` dictionary with a suitable string as a key. This will automatically add it to the output of the `pylie.methods()` function that lists all implemented time steppers.

```python
# hmanifold/mymanifold.py

import numpy as np
from .hmanifold import HomogenousManifold
from ..liegroup import MyLieGroup
from ..liealgebra import MyLieAlgebra


class MyManifold(HomogenousManifold):
    """
    Text describing the manifold, e.g. any restrictions
    or its associated Lie group. Will appear in the output
    of pylie.manifolds().
    """
    # Make sure to pick a suitable initial value for y
    def __init__(self, y=np.array([0, 0, 1])):
        self.n = y.size
        self.y = y
        self.lie_group = MyLieGroup()
        self.lie_algera = MyLieAlgebra()

    # If desired, y can be checked against various constraints
    # If so, the two followin methods must be implemented with the
    # @decorators as indicated.
    # This is optional.

    @property
    def y(self):
        return self._y

    @y.setter
    def y(self, value):
        if not some_condition:
            raise ValueError(
                "This is not a valid element of MyManifold"
              )
        self._y = value
```

Listing 3: An example of how to implement a new manifold in PyLie.

```python
# liegroup/liegroup.py

class MyLieGroup(LieGroup):
    def action(self, g, u):
        # Implement the action here
        return result

    # Auxiliary methods may optionally be defined below
```

Listing 4: The minimum requirements to a definition of a Lie group class in PyLie.

```python
# liealgebra/liealgebra.py

class MyLieAlgebra(LieAlgebra):
    def __init__(self, LieGroup):
        # This binds the action of the corresponding
        # to a self.action() method for use in exp and dexp,
        # if necessary
        super().__init__(LieGroup)

    def exp(self, y):
        # If y is a matrix, this method may be left undefined
        # to use the scipy implementation of the matrix
        # exponential instead
        return result

    def dexpinv(self, u, v, q: int):
        # q denotes the order of dexp^{-1}_u (v)
        # Again, this may be left undefined if
        # u and v are matrices and you would like
        # to use a truncated series with the matrix
        # commutator.
        return result
```

Listing 5: An example of how to implement a Lie algebra in PyLie. Note that if the elements of the algebra are matrices, the base `LieAlgebra` class may be used insted.

```
# hmanifold/__init__.py

# Previously existing imports above
from .mymanifold import MyManifold

# Assuming __all__ is defined above
__all__ += ["MyManifold"]
```

Listing 6: Exposing a function to the rest of PyLie.

```
# timestepper/timestepper.py

class Heuns(TimeStepper):
    def __init__(self, manifold):
        super().__init__(manifold)
        self.a = np.array([
            [0, 0],
            [1, 0]
        ])
        self.b = np.array([0.5, 0.5])
        self.c = np.array([0, 1])
        self.order = 2
        self.s = 2
```

Listing 7: An implementation of Heun's method in PyLie.

```python
# timestepper/timestepper.py

class CustomSolver(TimeStepper):
    """
    Text describing the method. Will be included in the output of
    pylie.methods()
    """
    def __init__(self, manifold):
        # To make exp, dexpinv and action available:
        super().__init__(manifold)

    def step(self, f, t, y, h):
        # Calculate the next step here
        return result
```

Listing 8: Example of how to implement a custom solver that does not utilize the RKMK framework.

Chapter 4

Numerical examples

In this chapter, we will present three different mechanical systems. We describe their dynamics, and we apply Runge-Kutte Munthe-Kaas methods to the resulting differential equations.

For each problem, we also calculate a reference solution using the RK45 integrator from SciPy with a low error tolerance [8, 39]. This solution is used to approximate the error of the applied RKMK methods as follows: Each problem is solved in the timespan $[0, T]$ for some $T > 0$ using different step sizes h_i. T The reference solution is then used to calculate the corresponding global error e_{h_i}. By plotting this error against the step size used in a logarithmically scaled plot, a global error on the form (1.2) will appear as a straight line with slope p.

We will also inspect the geometric properties of the numerical solutions and compare these to a solution obtained with the RK45 method as implemented in SciPy, used with the default accuracy parameters.

4.1 Rigid Body Equations

The rigid body equations, also known as the Euler equations, describe the rotational motion of a body that is not acted upon by any forces and has some given initial motion. The equations describe the angular velocity of the body as a function of time, denoted $\omega(t)$, relative to a coordinate system fixed to the rigid body—see Figure 4.1.

As the body is not acted upon by any forces, the magnitude of the angular velocity will remain constant. Thus, after scaling the equation, $\omega(t) \in S^2$ for all t, and so the equations may be expressed through the action of SO(3). Representing elements of SO(3) as orthogonal 3-by-3 matrices, it can be shown from the definition of the Lie algebra $\mathfrak{so}(3) = T_e\mathrm{SO}(3)$ that its elements must be 3-by-3 skew-symmetric matrices. To see this, let $\gamma(t)$ be a curve in SO(3) such that $\gamma(0) = e = I$, with I being the identity matrix. We have

$$\gamma(t)^T \cdot \gamma(t) = I,$$

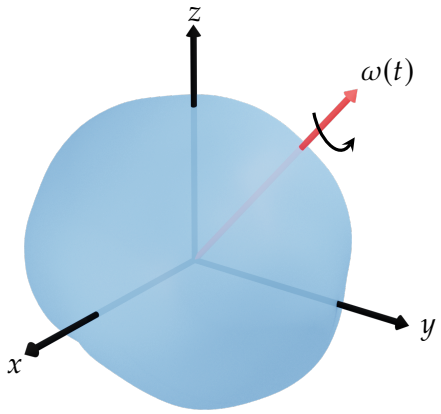


Figure 4.1: An illustration of a rigid body with angular momentum $\omega(t)$, the quantity of interest in the rigid body equations.

and so differentiating with respect to time at $t = 0$ yields

$$\dot{\gamma}(0)^T + \dot{\gamma}(0) = 0.$$

The hat map $\hat{\cdot} : \mathbb{R}^3 \to \mathfrak{so}(3)$ is a useful construct that lets us represent elements of $\mathfrak{so}(3)$ as vectors in $\mathbb{R}^3$ by the map

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \mapsto \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix}.$$

Note that for $x, y \in \mathbb{R}^3$, $\hat{x}y = x \times y$ where $\times : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3$ is the usual cross-product. If the body coordinate system is placed along the principal axes of the rigid body, the equations of motions are given by

$$\dot{\omega}(t) = \begin{bmatrix} 0 & \omega_3/J_3 & -\omega_2/J_2 \\ -\omega_3/J_3 & 0 & \omega_1/J_1 \\ \omega_2/J_2 & -\omega_1/J_1 & 0 \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} \tag{4.1}$$

where J_k are the moments of intertia of the corresponding axes. A derivation of the equations may be found in e.g. [18]. The equation is already on the form of (2.5), so we may apply the RKMK methods directly with

$$f(\omega) = -\widehat{J^{-1}\omega}. \tag{4.2}$$

Before moving on to the numerical results, we state the closed formulas for the maps $\exp : \mathfrak{so}(3) \to \mathrm{SO}(3)$ and $\mathrm{dexp}_u^{-1} : \mathfrak{so}(3) \to \mathfrak{so}(3)$.

The exponential map in $\mathfrak{so}(3)$

Let $x \in \mathbb{R}^3$ such that $\hat{x} \in \mathfrak{so}(3)$ and define $\alpha := \|x\|_2$. We have that $\hat{x}^3 = -\alpha^2 \hat{x}$, and so

$$\hat{x}^{2k+1} = (-1)^k \alpha^{2k} \hat{x} \qquad \text{for } k = 0, 1, \ldots.$$

By writing the exponential as

$$\exp \hat{x} = \sum_{n=0}^{\infty} \frac{\hat{x}^n}{n!} = \sum_{n=0}^{\infty} \frac{\hat{x}^{2n}}{(2n)!} + \sum_{n=0}^{\infty} \frac{\hat{x}^{2n+1}}{(2n+1)!} \tag{4.3}$$

we can solve for the odd and even terms separately to find

$$\sum_{n=0}^{\infty} \frac{\hat{x}^{2n+1}}{(2n+1)!} = \hat{x} \sum_{n=0}^{\infty} \frac{(-1)^n \alpha^{2n}}{(2n+1)} = \frac{\sin \alpha}{\alpha} \hat{x}, \tag{4.4}$$

$$\begin{aligned} \sum_{n=0}^{\infty} \frac{\hat{x}^{2n}}{(2n)!} = I + \hat{x} \sum_{n=0}^{\infty} \frac{\hat{x}^{2n+1}}{(2n+2)!} &= I + \hat{x}^2 \sum_{n=0}^{\infty} \frac{(-1)^n \alpha^{2n}}{(2n+2)!} \\ &= I + \frac{\hat{x}^2}{\alpha^2} \left(1 - \sum_{n=0}^{\infty} \frac{(-1)^n \alpha^{2n}}{(2n)!} \right) = I + \frac{1 - \cos \alpha}{\alpha^2} \hat{x}^2. \end{aligned} \tag{4.5}$$

Thus, $\exp \colon \mathfrak{so}(3) \to \mathrm{SO}(3)$ is given by

$$\exp \hat{x} = I + \frac{\sin \alpha}{\alpha} \hat{x} + \frac{1 - \cos \alpha}{\alpha^2} \hat{x}^2, \tag{4.6}$$

known as Rodrigues' formula [2].

The expression for dexp^{-1} may be found in a similar manner. First note that for $x, y, z \in \mathbb{R}^3$,

$$[\hat{x}, \hat{y}] z = x \times (y \times z) - y \times (x \times z),$$

and so by the Jacobi identity,

$$[\hat{x}, \hat{y}] = \widehat{(x \times y)}.$$

Identifying elements of $\mathfrak{so}(3)$ with elements in $\mathbb{R}^3$, this implies

$$\mathrm{ad}_x(y) = \hat{x} y,$$

which again means

$$f(\mathrm{ad}_x) y = f(\hat{x}) y.$$

From this, it is possible to show that

$$\mathrm{dexp}_x^{-1} = \frac{\hat{x}}{\exp \hat{x} - I} = I - \frac{1}{2} \hat{x} - \frac{2 - \alpha \cot(\alpha/2)}{2\alpha^2} \hat{x}^2,$$

see [3].

Numerical results

Equation (4.2) was solved with a variety of step sizes in order to approximate the error of Explicit Euler, Heun's method and Runge-Kutta Munthe-Kaas 4. For each step length, the solution was compared to a SciPy solution of (4.1) at time $t = 30$. We used $\omega(0) = [\cos(1.1), 0, \sin(1.1)]^T$ and $J = \text{diag}(2.2, 1.0, 2.3)$. The results, plotted in Figure 4.2, indicate that the methods perform as one would expect. For small step sizes, the global error of RKMK4 seems to reach a minimum and stops improving. This is due to the fact that it reaches an accuracy comparable to that of the reference solution.

As $\omega \in S^2$, we expect all methods to satisfy $\omega_n^T \omega_n = 1$ for all t_n. We test this by using a step length of $h = 0.5$ to compute the solution in the timespan $[0, 1000]$, i.e. $t_n = n/2$ for $n = 0, 1, \ldots, 2000$, and then calculate $1 - \omega_n^T \omega_n$ at each step. The results, shown in Figure 4.3, demonstrate that for all RKMK methods $\omega_n^T \omega_n - 1$ is of order 10^{-15}, i.e. the constraint is satisfied to within machine precision [5]. For reference, we also perform the same experiment with the RK45 method of SciPy, letting the algorithm decide its own step length based on the standard error tolerance parameters. Here $\omega_n^T \omega_n - 1$ is seen to be of order 10^{-2}.

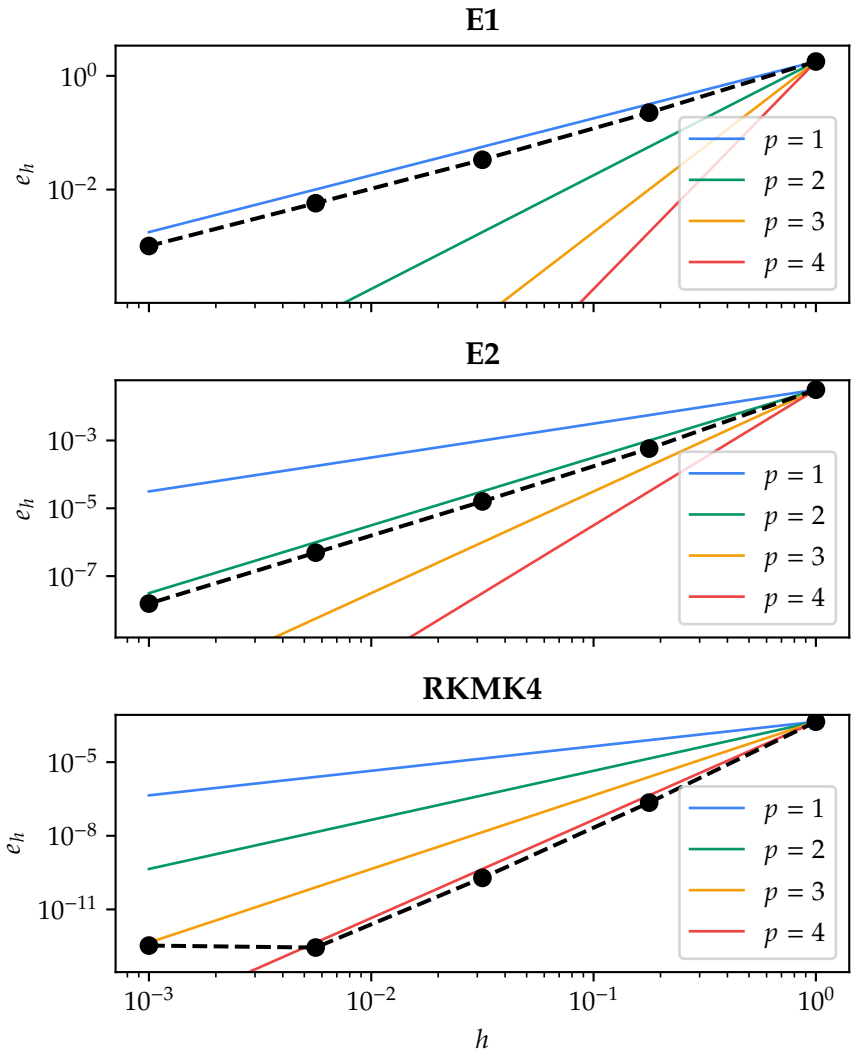


Figure 4.2: Approximation of the global error of the PyLie implementations of Explicit Euler (E1), Heun's method (E2) and RKMK4 when applied to the rigid body equations. The reference solution was computed with SciPy. The coloured lines are polynomials with a single term of order p. For the shortest step lengths, RKMK4 matches the accuracy of the reference solution.

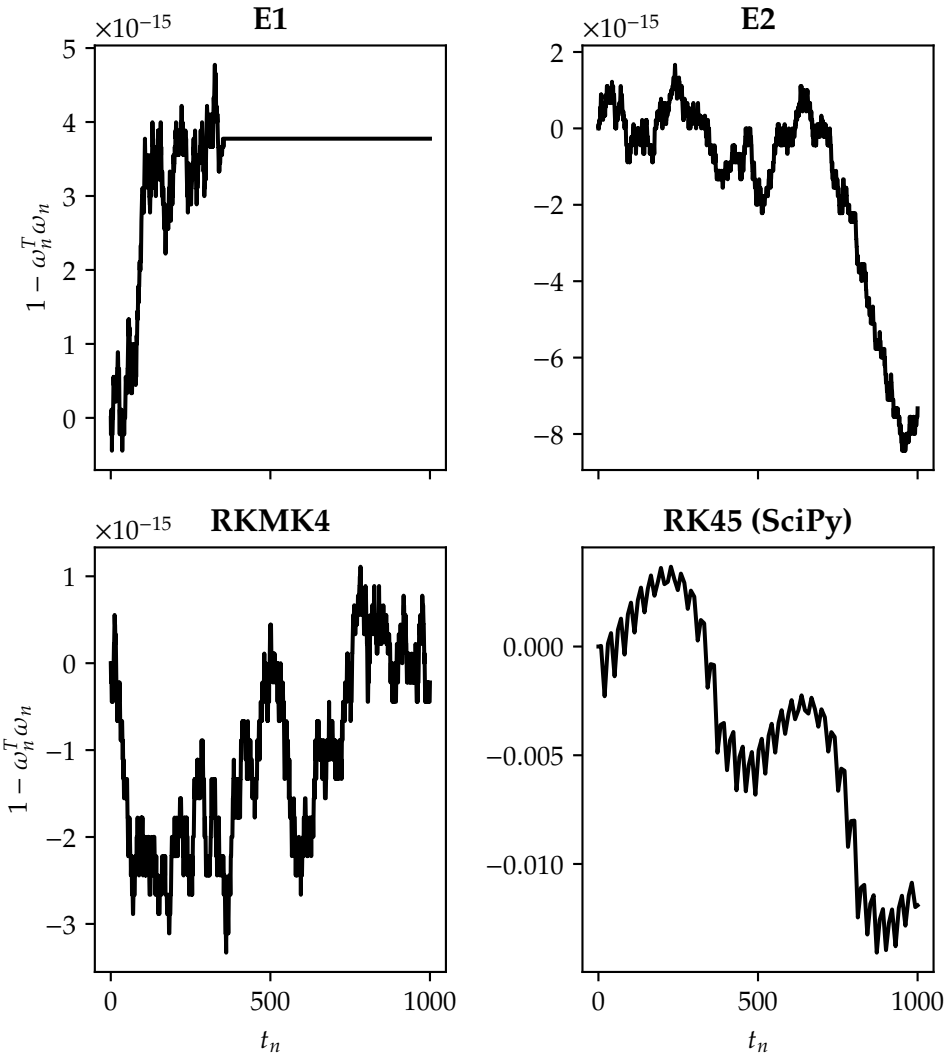


Figure 4.3: The value of $1-\omega_n^T\omega_n$ at each time step for different methods with ω_n being a numerical solution to the rigid body eqations. This value should equal 0.

4.2 Heavy Top Equations

The heavy top equations describe the dynamics of a rigid body rotating about a supported fixed point in a constant gravitational field. In [21], the equations are given as

$$\begin{aligned} \dot{\Omega} &= \Omega \times J^{-1}\Omega + mg\Gamma \times \chi, \\ \dot{\Gamma} &= \Gamma \times J^{-1}\Omega, \end{aligned} \tag{4.7}$$

where

- $\Omega \in \mathbb{R}^3$ is the body angular momentum vector,
- $J = \text{diag}(J_1, J_2, J_3)$ is the moment of intertia tensor,
- $\Gamma = R^T(t)e_3$ represents the motion of the unit vector along the vertical axis as seen from the rotating body,
- χ is the constant vector in the body reference point of frame, going from the point of support to the body's centre of mass,
- m is the mass of the body, and finally,
- g is the acceleration due to gravity.

A simple heavy top is illustrated in Figure 4.4.

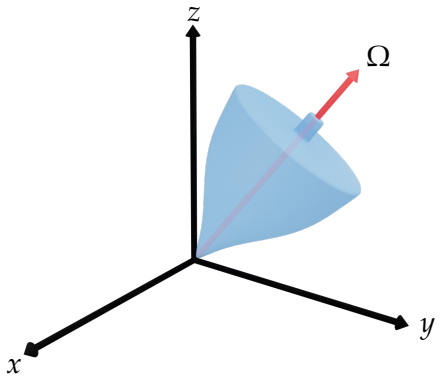


Figure 4.4: A simple heavy top with the fixed point in the origin of the global reference frame.

Even though the spinning top is a rigid body, the presence of gravitational forces means the equations cannot be formulated with the action of SO(3). In [7] it is shown that the system may be formulated on the differentiable manifold $\mathfrak{se}(3)^*$, the dual of the Lie algebra of SE(3). Elements of SO(3) will be represented as pairs (g, u) with $g \in \text{SO}(3)$ and $u \in \mathbb{R}^3$. Elements of both $\mathfrak{se}(3)$

and $\mathfrak{se}(3)^*$ will be represented as vectors in $\mathbb{R}^3 \times \mathbb{R}^3$. The left action of SE(3) on $\mathfrak{se}(3)^*$ is given by

$$\Lambda\left((g,u),(x,y)\right) = (g(x + u \times gv), gv),$$

with infinitesimal generator

$$\lambda_*|_{(\Omega,\Gamma)}\left((x,y)\right) = (x \times \Omega + y \times \Gamma, x \times \Gamma).$$

The heavy top equations (4.7) may then be written

$$(\dot{\Omega}, \dot{\Gamma}) = \lambda_*|_{(\Omega,\Gamma)}\left((-J^{-1}\Omega, -mg\chi)\right),$$

so that with reference to (2.5) we get

$$f(y) = f(\Omega,\Gamma) = (-J^{-1}\Omega, -mg\chi).$$

The exponential map in $\mathfrak{se}(3)$

In [7] the exponential map $\exp\colon \mathfrak{se}(3) \to \mathrm{SE}(3)$ is given as

$$\exp(x,y) = \left(\exp\hat{x}, \frac{\exp\hat{x} - I}{\hat{x}} y\right).$$

The $\exp\hat{x}$ mapping given on the right-hand side of the equation is the exponential map in $\mathfrak{so}(3)$, given in (4.6). The fraction in the second component may not be computed directly, as the matrix $\hat{x}$ is singular.[1] Its Taylor series is given by

$$\frac{\exp\hat{x} - I}{\hat{x}} = \sum_{n=0}^{\infty} \frac{1}{(n+1)!}\hat{x}^n.$$

Using a similar procedure as in equations (4.3)–(4.5) one can show that

$$\frac{\exp\hat{x} - I}{\hat{x}} = I + \frac{1 - \cos\alpha}{\alpha^2}\hat{x} + \frac{\alpha - \sin\alpha}{\alpha^3}\hat{x}^2,$$

where again $\alpha = \|x\|_2$.

To derive an expression for $\mathrm{dexp}^{-1}_{(x,y)}$, we present an extended discussion of a result first shown in [33], later published in [4]. With the Lie bracket in $\mathfrak{se}(3)$ given in [7] as

$$[(x,y),(u,v)] = (x \times u, x \times v - u \times y),$$

the $\mathrm{ad}_{(x,y)}$ operator may be expressed as the 6×6 block matrix

$$\mathrm{ad}_{(x,y)} = \begin{bmatrix} \hat{x} & 0 \\ \hat{y} & \hat{x} \end{bmatrix}.$$

[1] Note e.g. that $\hat{x} \cdot x = 0$.

By the Caley-Hamilton theorem [36],

$$\mathrm{ad}_{(x,y)}^{2k+1} = \begin{bmatrix} (-1)^k \alpha^{2k} \hat{x} & 0 \\ (-1)^k (\alpha^{2k} \hat{y} + \alpha^{2(k-1)} 2k x^T y \hat{x}) & (-1)^k \alpha 2k \hat{x} \end{bmatrix}.$$

Given a function $f(z)$ that is real analytic at $z = 0$, i.e. $f(z) = \sum_{n=0}^{\infty} f_n z^n$ with real coefficients f_n, we define

$$f_+(z) = \frac{1}{2}(f(iz) + f(-iz)) = \sum_{n=0}^{\infty} (-1)^n f_{2n} z^{2n},$$
$$f_-(z) = \frac{1}{2}(f(iz) - f(-iz)) = \sum_{n=0}^{\infty} (-1)^n f_{2n+1} z^{2n+1}.$$

It can then be shown that

$$f(\mathrm{ad}_{(x,y)}) = f_0 I + \begin{bmatrix} g_1(\alpha)\hat{x} & 0 \\ g_1(\alpha)\hat{y} + \tilde{g}_1(\alpha)\hat{x} & g_1(\alpha)\hat{x} \end{bmatrix} + \begin{bmatrix} \hat{x} & 0 \\ \hat{y} & \hat{y} \end{bmatrix} \begin{bmatrix} g_2(\alpha)\hat{x} & 0 \\ g_2(\alpha)\hat{y} + \tilde{g}_2(\alpha)\hat{x} & g_2(\alpha)\hat{x} \end{bmatrix},$$

where

$$g_1(\alpha) = \frac{f_-(\alpha)}{\alpha},$$
$$\tilde{g}_1(\alpha) = \frac{\rho}{\alpha} \frac{\mathrm{d}}{\mathrm{d}\alpha} g_1(\alpha),$$
$$g_2(\alpha) = \frac{f_0 - f_+(\alpha)}{\alpha},$$
$$\tilde{g}_2(\alpha) = \frac{\rho}{\alpha} \frac{\mathrm{d}}{\mathrm{d}\alpha} g_2(\alpha)$$

with $\rho = x^T y$. Writing this out component by component, one finds that $f(\mathrm{ad}_{(x,y)})(u, v) = (\eta, \xi)$ where

$$\eta = f_0 u + g_1(\alpha) x \times (x \times u)$$

and

$$\xi = f_0 v + g_1(\alpha)(y \times u + x \times v) + \tilde{g}_1(\alpha) x \times u + \tilde{g}_2(\alpha) x \times (x \times u) + g_2(\alpha)(y \times (x \times u) + x \times (y \times u) + x \times (x \times v)).$$

In the special case of $f(\mathrm{ad}_{(x,y)}) = \mathrm{dexp}^{-1}_{(x,y)}$, we get

$$\begin{aligned} f_0 &= 1 \\ g_1(\alpha) &= -\frac{1}{2}, \\ \tilde{g}_1(\alpha) &= 0, \\ g_2(\alpha) &= \frac{1 - \frac{\alpha}{2}\cot\frac{\alpha}{2}}{\alpha^2}, \\ \tilde{g}_2(\alpha) &= \frac{\rho\left(\alpha^2 \csc^2\frac{\alpha}{2} + 2\alpha\cot\frac{\alpha}{2} - 8\right)}{4\alpha^4}. \end{aligned}$$

These expressions are implemented in the class `se3LieAlgebra` class of PyLie.

Numerical results

We repeat the numerical order analysis of the methods on the rigid body equations. The parameters used were $J = \mathrm{diag}(2, 2, 1)$, $T = 10$ and $y_0 = [0.2, 0.3, 1.0, \cos(0.2), \sin(0.2), 0]^T$. The results shown in Figure 4.5 show that the methods perform as expected. However, it is noteworthy that the step errors need to be below a certain threshold before the global error of Lie-Euler is reduced. This is not surprising: Recall that (2.6) only holds for small t, and as we go too far, the solution is no longer valid. It is also intuitive that taking very long steps may miss important regions in the vector field defined by the ODE. Again, the global error of RKMK4 flattens out as the method reaches an estimated accuracy similar to that of the reference solution.

According to [9], the heavy top equations have four conserved quantities, of which we will inspect two: The projection of the angular momentum on the body vertical unit vector $\Omega \cdot \Gamma$, and the norm $\|\Gamma\|_2$. Results are plotted in Figures 4.6 and 4.7. Again, note that the Munthe-Kaas methods preserve the invariants to machine precision, in contrast to the standard RK45 method from SciPy.

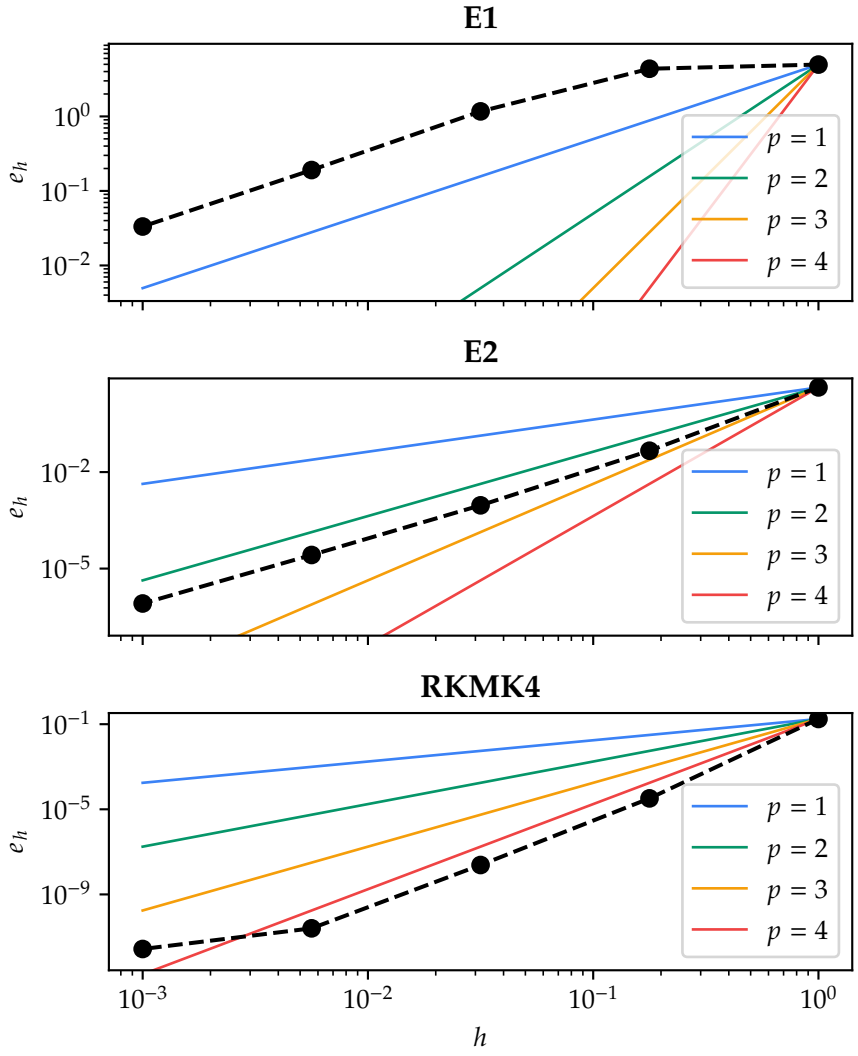


Figure 4.5: Order approximations of the PyLie implementations of Explicit Euler (E1), Heun's method (E2) and RKMK4 when applied to the heavy top equations. The coloured lines are polynomials with a single term of order p. The reference solution used to compute the global error was obtained with SciPy.

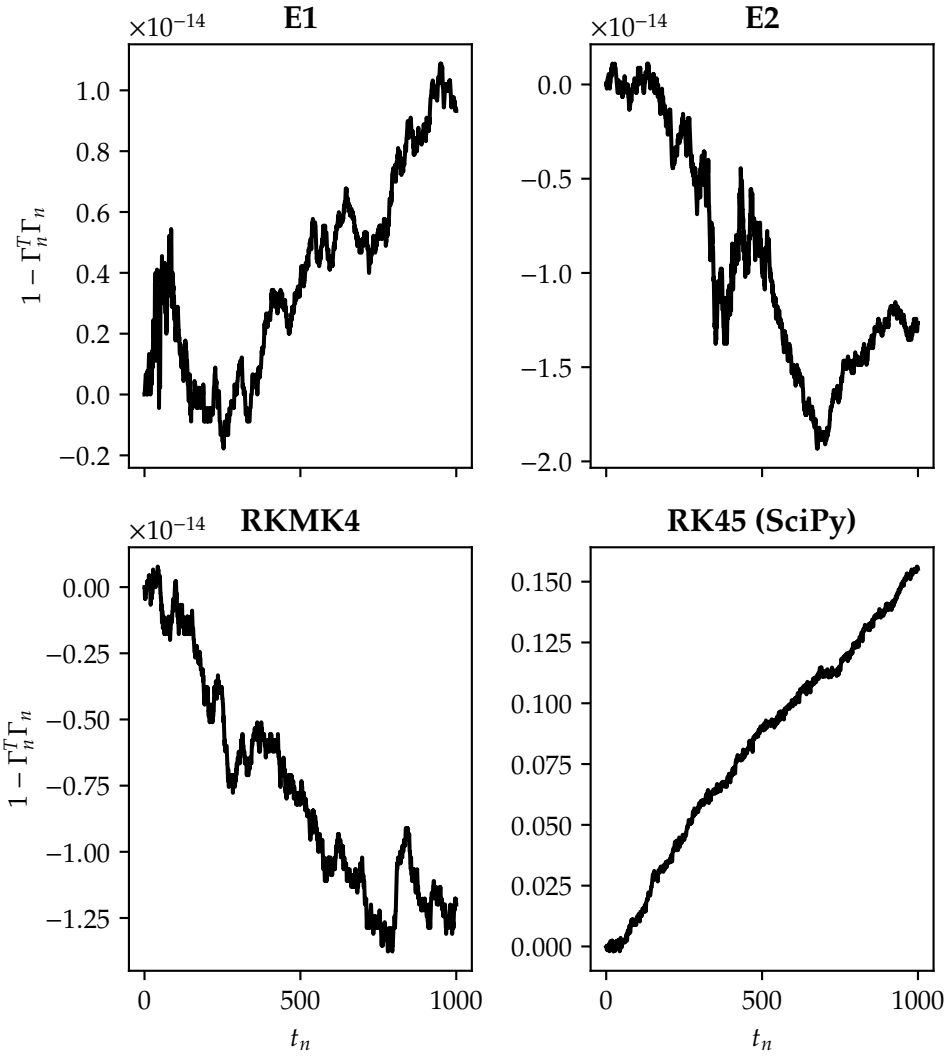


Figure 4.6: Conservation of the invariant $\Gamma^T\Gamma$ by the different methods as a function of t_n.

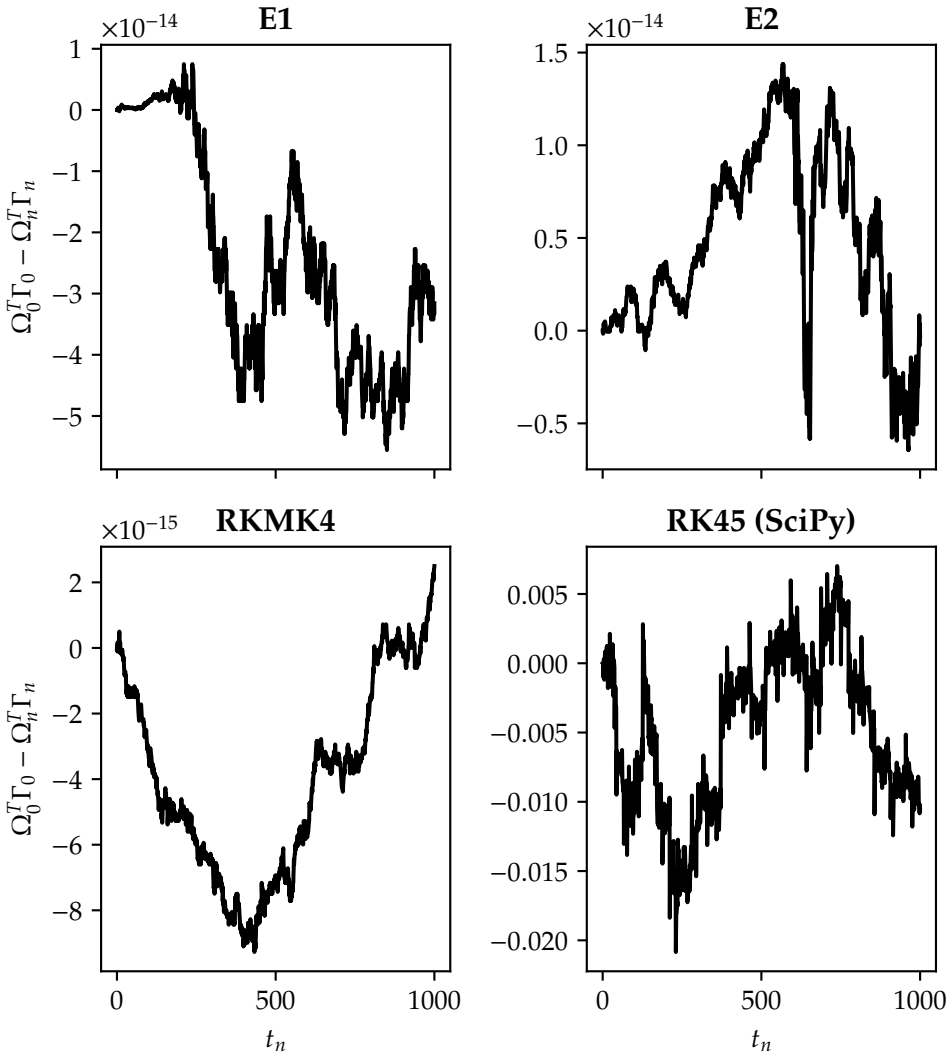


Figure 4.7: Conservation of the invariant $\Omega^T\Gamma$ as a function of t_n.

4.3 The Chained Spherical Pendulum

We consider a system of pendulums linked together in the following manner: Pendulum i of mass m_i is connected to pendulum $i-1$ with a rod of length ℓ_i for $i = 2, 3, \ldots, N$. The first pendulum of mass m_1 is connected to a fixed point in the origin by a rod of length ℓ_1. The system is illustrated in Figure 4.8.

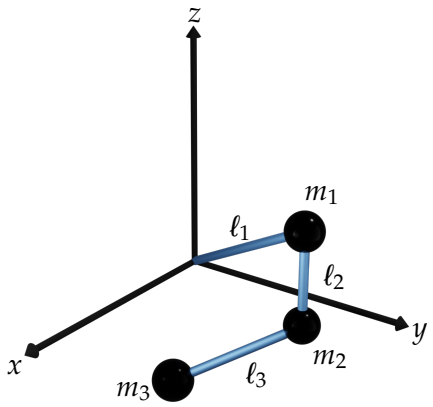


Figure 4.8: The system of chained spherical pendulums with $N = 3$.

The description of the system and its definition through a Lie group action is based upon [4], which in turn builds upon [26]. The model is not entirely realistic: For instance, the interaction between the movements of the pendulums is not taken into account. It is also assumend the each pendulum may rotate freely, ignoring any collisions that would occur. Nonetheless, it serves as an instructive example for the application of Lie group integrators. By increasing N and at the same time ensuring $\sum_{i=1}^{N} \ell_i = \textit{constant}$, it is also concievable that the equations may model e.g. a beam and be of interest in practical applications.

The pendulum is described in terms of the positions $q_i \in S^2 \subset \mathbb{R}^3$ of pendulum $i = 1, 2, \ldots, N$, and their angular velocities $\omega_i \in T_{q_i}S^2$. Note that

$$T_{q_i}S^2 = \{v \in \mathbb{R}^3 : v^T q_i = 0\} \subset \mathbb{R}^3,$$

and so for any velocity $\dot{q}_i \in T_{q_i}S^2$ there is some $\omega_i \in \mathbb{R}^3$ such that $\dot{q}_i = \omega_i \times q_i$ with ω_i being the angular velocity of q_i. This justifies the assumption that $\omega_i \in T_{q_i}S^2$[2], and the dynamics of the system may be described by the coordinates

$$y = (q_1, \omega_1, q_2, \omega_2, \ldots, q_N, \omega_N) \in (TS^2)^N.$$

For simplicity, we will use the notation $y = (q, \omega)$ with $q = (q_1, \ldots, q_N)$ and $\omega = (\omega_1, \ldots, \omega_N)$.

[2] the alternative being that ω_i is parallell to q_i, in which case $\omega_i = \dot{q}_i = 0$.

To describe the dynamics with an infiniteismal generator, we begin with the action of the Lie group SE(3) on its Lie algebra $\mathfrak{se}(3)$. This is denoted $\mathrm{Ad}\colon \mathrm{SE}(3) \times \mathfrak{se}(3) \to \mathfrak{se}(3)$, and is given by

$$\mathrm{Ad}\big((g,u),(x,y)\big) = (gx, gy + \hat{u}gx).$$

Using the identification of elements of $\mathfrak{se}(3)$ as vectors $x \in \mathbb{R}^3 \times \mathbb{R}^3$, we may define the action $\Lambda\colon \mathrm{SE}(3) \times \mathbb{R}^6 \to \mathbb{R}^6$ by

$$\Lambda\big((g,u),(x,y)\big) = (gx, gy + \hat{u}gx).$$

We see that if $(x,y) \in TS^2_{|q|}$, where

$$TS^2_{|q|} = \big\{(x,y) \in \mathbb{R}^3 \times \mathbb{R}^3 \colon y^T x = 0, |x| = |q|\big\} \subset \mathbb{R}^6,$$

we may consider Λ to be an action $\Lambda\colon \mathrm{SE}(3) \times TS^2_{|q|} \to TS^2_{|q|}$. It is also possible to show that for a point $m \in TS^2_{|q|}$, the orbit $O(m) = TS^2_{|q|}$ [32]. Thus, with $|q| = 1$, Λ defines a transitive Lie group action on TS^2. The infinitesimal generator of this action is given by

$$\lambda_*((x,y))\big|_{(q,\omega)} = (\hat{x}q, \hat{x}\omega + \hat{y}q).$$

The necesary action to describe a pendulum chain with $N > 1$ is obtained through the group action of the Lie group defined by the N-times cartesian product of SE(3). That is, with $\cdot$ being the group product of SE(3) and elements $g_1 = (g_1^{(1)}, \ldots, g_1^{(N)}), g_2 = (g_2^{(1)}, \ldots, g_2^{(N)}) \in (\mathrm{SE}(3))^N$, the group product $\circ$ on $(SE(3))^N$ is given by

$$g_1 \circ g_2 = (g_1^{(1)} \cdot g_2^{(1)}, \ldots, g_1^{(N)} \cdot g_2^{(N)}).$$

This yields the action $\Lambda\colon (\mathrm{SE}(3))^N \times (TS^2)^N \to (TS^2)^N$ defined by

$$\begin{aligned}&\Lambda((g_1,u_1,\ldots,g_N,u_N),(q_1,\omega_1,\ldots,q_N,\omega_N))\\ &= (g_1q_1, g_1\omega_1 + \hat{u}_1 g_1 q_1, \ldots, g_N q_N, g_N\omega_N + \hat{u}_N g_N q_N).\end{aligned}$$

The infinitesimal generator of this action is given by

$$\lambda_*|_m(\xi) = (\hat{x}_1 q_1, \hat{x}_1\omega_1 + \hat{y}_1 q_1, \ldots, \hat{x}_N q_N, \hat{x}_N\omega_N + \hat{y}_N q_N)$$

where $m = (q_1,\omega_1,\ldots,q_N,\omega_N) \in (TS^2)^N$ and $\xi = (x_1,y_1,\ldots,x_N,y_N) \in (\mathfrak{se}(3))^N$.

In [26], the Lagrangian of the chain of pendulums is stated as

$$\begin{aligned}\mathcal{L}(q,\omega) &= T(q,\omega) - U(q)\\ &= \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} M_{ij}\omega_i^T \hat{q}_i^T \hat{q}_j \omega_j - \sum_{i=1}^{N}\left(\sum_{j=i}^{N} m_j\right) g\ell_i e_3^T q_i,\end{aligned}$$

where

$$M_{ij} = \left(\sum_{k=\max(i,j)}^{N} m_k \right) \ell_i \ell_j I \in \mathbb{R}^{3\times 3}$$

is the inertia tensor of the system. Note that $\hat{q}_i^T \hat{q}_i = I - q_i q_i^T$, and so $\omega_i^T \hat{q}_i^T \hat{q}_i \omega_i = \omega_i^T \omega_i$. This means that we may simplify the Lagrangian $\mathcal{L}$ by defining $R(q)$ to be the $3N$-by-$3N$ symmetric block matrix defined by

$$R(q)_{ij} = \begin{cases} \left(\sum_{k=i}^{N} m_k \right) \ell_i^2 I & \text{if } i = j, \\ \left(\sum_{k=j}^{N} m_k \right) \ell_i \ell_j \hat{q}_i^T \hat{q}_j & \text{if } i < j, \\ \left(R(q)_{ji} \right)^T & \text{otherwise.} \end{cases} \tag{4.8}$$

The kinetic energy of the system may then be written as

$$T(q, \omega) = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \omega_i^T R(q)_{ij} \omega_j.$$

The dynamics of the system is then given by the ODE

$$\dot{q}_i = \omega_i \times q_i. \qquad i = 1, 2, \ldots, N, \tag{4.9}$$

$$R(q)\dot{\omega} = \left[\sum_{\substack{j=1 \\ j\neq i}}^{N} M_{ij} \omega_j^T \omega_j \hat{q}_i q_j - \left(\sum_{j=i}^{N} m_j \right) g \ell_i \hat{q}_i e_3 \right]_{i=1,2,\ldots,N}. \tag{4.10}$$

By inspecting (4.8) it can be seen that for any $q \in (S^2)^N$ and $\omega \in T_{q_1} S^2 \times \ldots \times T_{q_N} S^2$, $(R(q)\omega)_i \in T_{q_i} S^2$. Thus, the linear map $A_q : T_{q_1} S^2 \times \ldots \times T_{q_N} S^2 \to T_{q_1} S^2 \times \ldots \times T_{q_N} S^2$ given by

$$A_q(\omega) = R(q)\omega$$

is well-defined. Further, since $R(q)$ is positive-definite it follows that A_q is invertible, and thus we have

$$\dot{\omega} = A_q^{-1} \left(\begin{bmatrix} g_1 & \cdots & g_N \end{bmatrix}^T \right) = \begin{bmatrix} h_1(q, \omega) \\ \vdots \\ h_N(q, \omega) \end{bmatrix}$$

where g_i is given by the ith component of the right hand side of (4.10). By the above discussion we may write $h_i = a_i(q, \omega) \times q_i$ for $i = 1, \ldots, N$ with a_i being some map from $(TS^2)^N$ to $\mathbb{R}^3$. Letting $a_i(q, \omega) = q_i \times h_i(q, \omega)$ achieves

the desired result, and so the appropriate function $f\colon \mathcal{M} \to (\mathfrak{se}(3))^N$ with respect to (2.5) is

$$f(q,\omega) = \begin{bmatrix} \omega_1 \\ q_1 \times h_1 \\ \vdots \\ \omega_N \\ q_N \times h_N \end{bmatrix}.$$

For further details, see [4] which in addition to a more thorough derivation of the dynamics provide an explicit formula for the system with $N = 2$.

Numerical results

We repeat the order analysis in a similar fashion as for the rigid body equations and the heavy top equations. For these experiments we use a system with $N = 2$ pendulums. The effect of N will be examined in a later trial. For simplicity we set $m_i = \ell_i = 1$ for $i = 1, 2, \ldots, N$. We set $g = 9.81$.

A random vector $y_0 = (q_1, \omega_1, q_2, \omega_2)$ was generated such that $q_i^T q_i = 1$ and $\omega_i^T q_i = 0$ for $i = 1, 2$. This initial value was reused throughout the order calculations. Trials show that too large values for the step length provide unreliable results, and so we use smaller values for the step size than we did in the two previous experiments in order to approximate the order of the methods. The results are shown in Figure 4.9. In this case, we see that all methods need a step size $h < 10^{-1}$ in order to improve as we expect. Again, we see that the estimated global error of RKMK4 stops improving as it reaches the same level of accuracy as the reference solution.

We also investigate how well the numerical schemes conserve two invariants of the system: Namely, $q_i^T q_i$ and $\omega_i^T q$, which we expect to be 1 and 0, respectively. Due to the necessity of using smaller step sizes, we calculate numerical solutions in the timespan $[0, 50]$ with a step length of 0.005, resulting in 10 000 steps. As before, the SciPy solver is used with its default arguments, using an embedded pair of RK methods to select an appropriate step length at each iteration. The results are shown in Figures 4.10 and 4.11. The RKMK methods conserve the invariants to machine precision, clearly outperforming the SciPy-implementation of RK45 in this respect.

Finally, we inspect how the number of pendulums N affect the global error. The reference solution is computed with the SciPy implementation of DOP853, an explicit method of order eight [16]. To reduce fluctuations due to the use of a random intial value, the global error is averaged over eight different trials for each N. The results are shown in Figure 4.12. There does not seem to be any obvious connection between the number of pendulums N and the global error committed of the methods, unless the accuracy of the SciPy solution is affected in a similar manner. This seems unlikely.

The motion of a pendulum with $N = 3$ is illustrated in Figure 4.13. As expected, the system exhibits chaotic behaviour as described in [41].

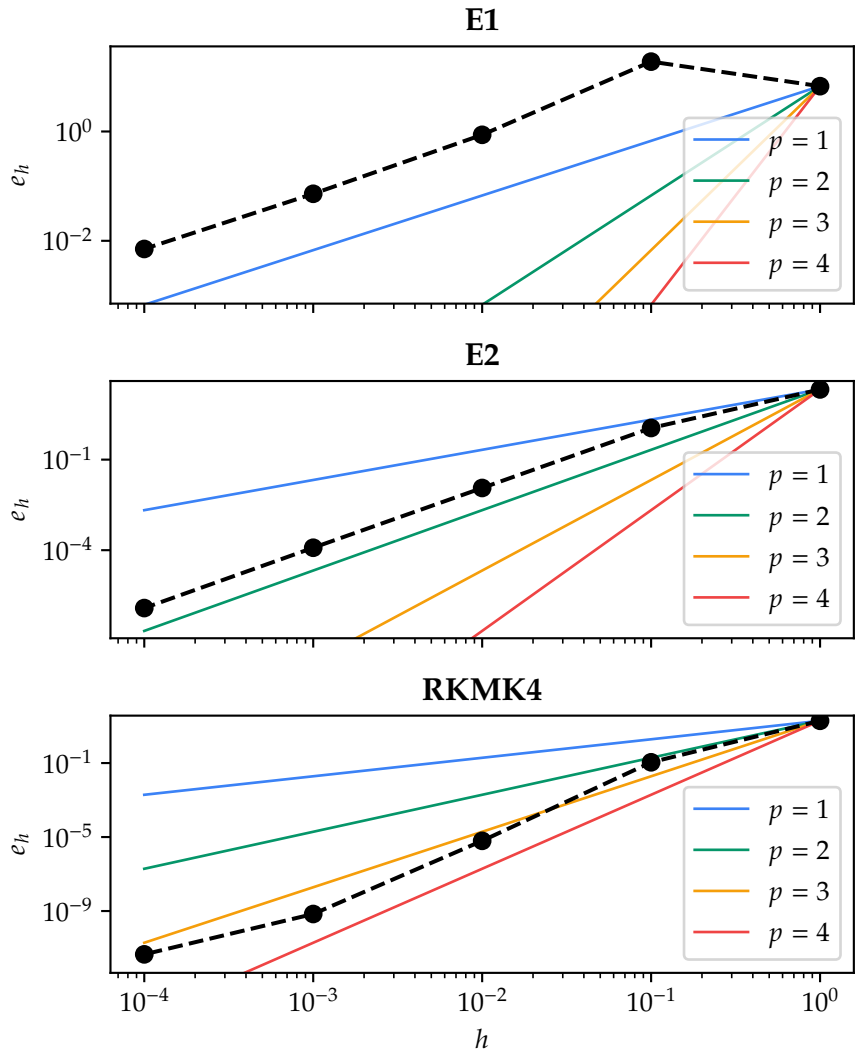


Figure 4.9: Order approximations of the PyLie implementations of Explicit Euler (E1), Heun's method (E2) and RKMK4 when applied to the chained spherical pendulum equations with $N = 2$. The coloured lines are polynomials with a single term of order p. The reference solution used to compute the global error was obtained with SciPy.

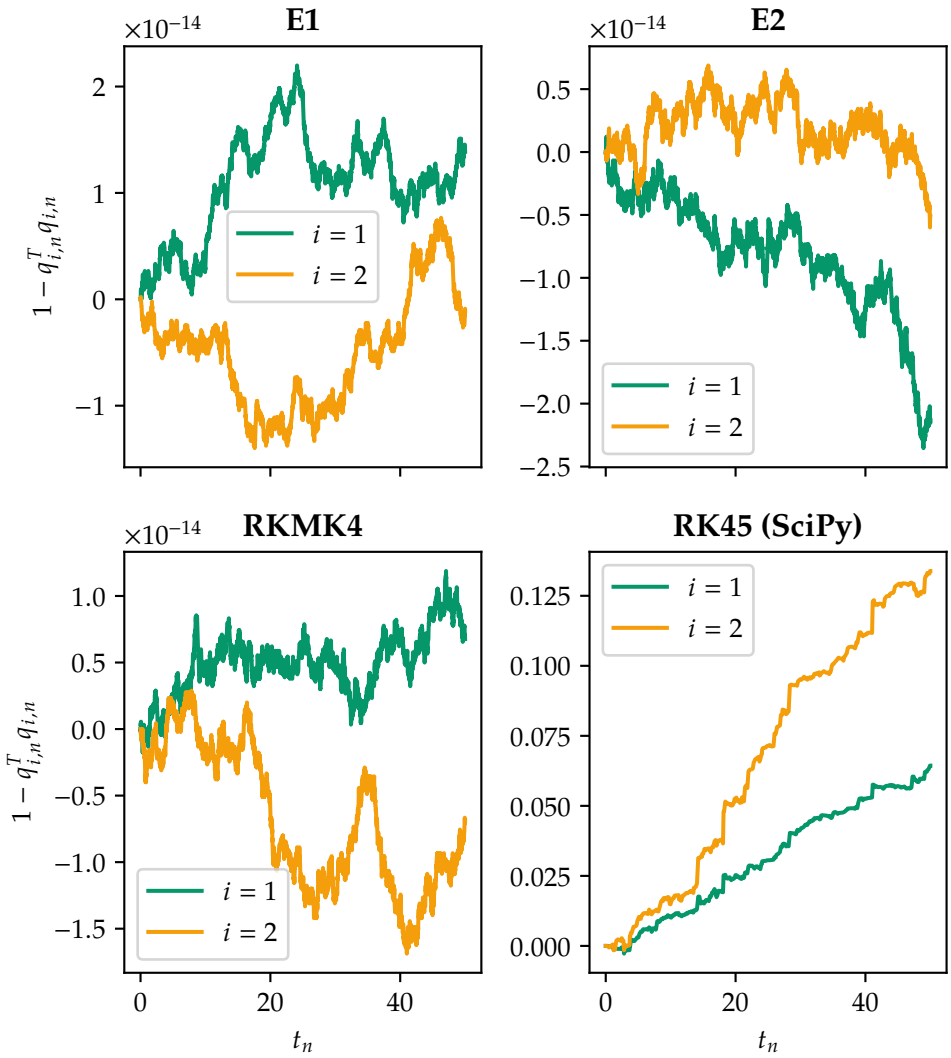


Figure 4.10: Conservation of the invariants $q_i^T q_i$ by the different methods as a function of t_n.

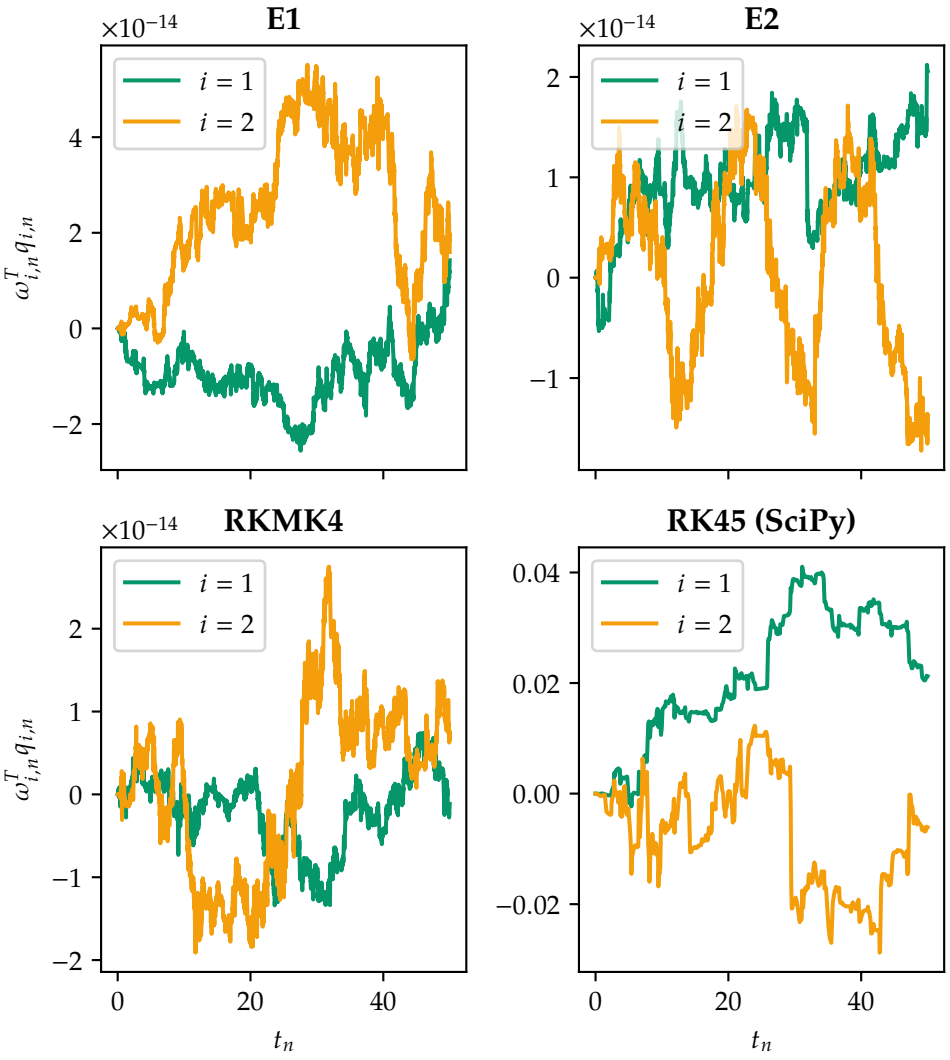


Figure 4.11: Conservation of the invariant $\omega_i^T q_i$ as a function of t_n.

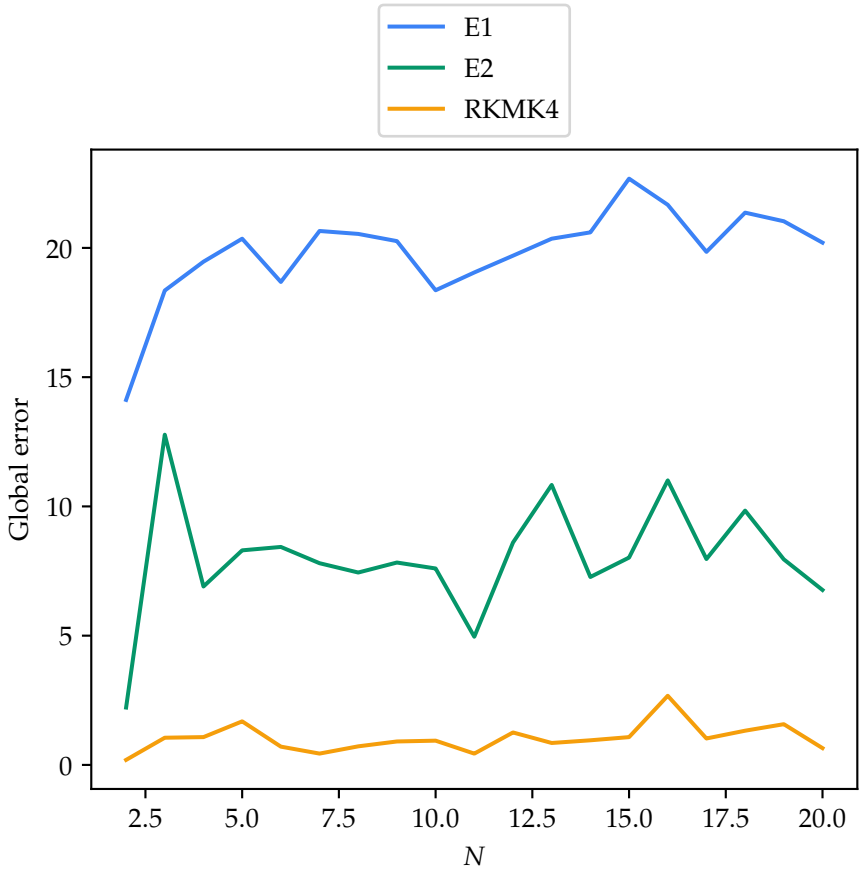


Figure 4.12: The global error of three differen numerical integrators from PyLie as a function of the number of pendulums N. For each N, the pendulum equations were solved for eight different random initial values and the global error was averaged. At each N the step length was $h = 0.1$, and the solution was found from $t_0 = 0$ to $t = 1$.

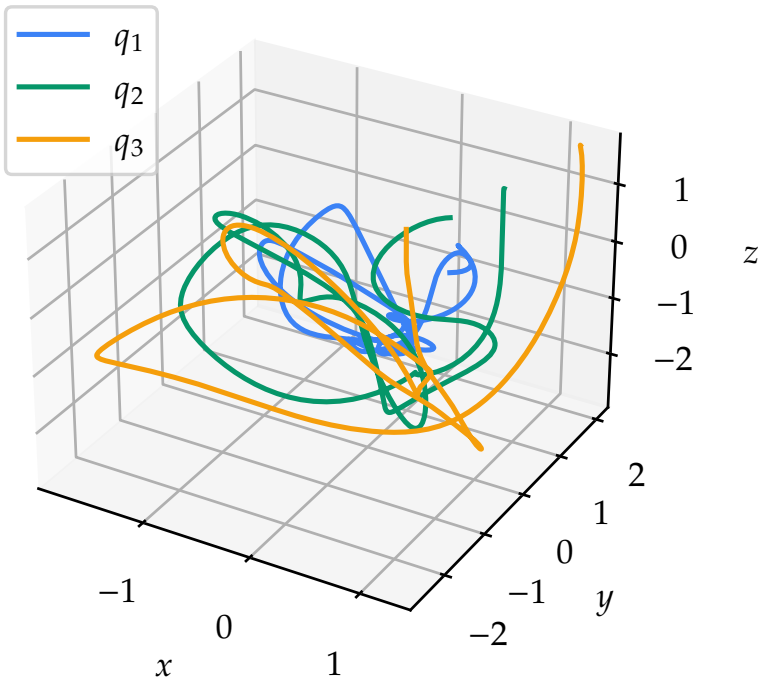


Figure 4.13: The motion of a pendulum with $N = 3$ in the time span $t \in [0, 5]$. The solution was obtained with the PyLie implementation of RKMK4 using a step length of $h = 0.01$.

Chapter 5

Conclusion

The main work of this thesis has been the development and testing of the Python package PyLie, a framework for solving differential equations evolving on non-linear manifolds. To this end, the Runge-Kutta Munthe-Kaas methods were motivated and introduced along with the basics of the underlying theory of manifolds, Lie groups, and Lie manifolds.

At the core of the methods are the mathematical constructs $\mathcal{M}$, G and $\mathfrak{g}$ and interactions between them, which must be represented in code in some meaningful way. In addition, the elements of each construct must be numerically represented in a suitable manner. To ensure consistency and correctness, it is natural to want to check each such representation against certain constraints depending on the mathematical object and the problem at hand.

Object-oriented programming is especially suitable for the task of implementing the methods. Representing e.g. a Lie algebra $\mathfrak{g}$ with a corresponding object `LieAlgebra` allows us to attach all maps relevant for $\mathfrak{g}$ as methods of `LieAlgebra`, and any programmatic object `x` may be verified to be a valid representation of an element $x \in \mathfrak{g}$.

In addition, inheritance makes extending the software fast and simple. As an example, consider the implementation of solving the chained spherical pendulum equations after finishing the heavy top equations. As both problems make use of the exp and dexp_u^{-1} maps with $\mathfrak{se}(3)$ as domain, the implementations of these maps were easily reused in a loop without having to define them a second time. This increases development speed and reduces the risk for programming errors.

The implementations of the methods were seen to satisfy what the Runge-Kutta Munthe-Kaas methods were designed to do: Namely, to conserve invariants and to ensure that the numerical solution y_n remains on the relevant manifold $\mathcal{M}$. However, this comes at a cost. The computational complexity of the methods is considerably higher when compared to the simpler Runge-Kutta methods. The RKMK methods also requires the differential equation

to be formulated in terms of the infinitesimal generator of the action of the relevant Lie group, a task that need not be trivial. Whether or not this extra computational and human effort is worth it will depend on the purpose of solving the problem: If the qualitative behavior of the numerical solution over a long time period is of great importance, the RKMK methods provide a practical way of achieving it.

Further work

Runge-Kutta Munthe-Kaas methods and the design of Python packages are both vast subjects, and this thesis has only scratched the surface of both. One obvious extension of PyLie would be to extend it to work for a greater number of problems, evolving on different manifolds and formulated using different Lie groups. Another is to increase the number of code tests in order to create even greater confidence in the correctness of the methods.

Another limitation is that all currently implemented methods are explicit. Implicit methods would be an interesting next step, but would require great care in how they are implemented. A possibly easier intermediary step would be to look beyond Runge-Kutta-coefficients to more modern developments. Examples include commutator-free methods and embedded pairs of coefficients reusing exponentials as explored in [7]. Implementing variable step-size methods in order to more efficiently find numerical solutions of a suitable accuracy is also highly relevant.

Bibliography

[1] Ralph Abraham, Jerrold E. Marsden, and Tudor Ratiu. *Manifolds, Tensor Analysis, and Applications*. Applied Mathematical Sciences. Springer, New York, NY, 1988. ISBN: 978-0-387-96790-5. DOI: 10.1007/978-1-4612-1029-0. URL: https://doi.org/10.1007/978-1-4612-1029-0.

[2] Elena Celledoni, Håkon Marthinsen, and Brynjulf Owren. "An introduction to Lie group integrators - basics, new developments and applications". In: *Journal of Computational Physics* 257, Part B (Jan. 2014), pp. 1040–1061. DOI: 10.1016/j.jcp.2012.12.031.

[3] Elena Celledoni and Brynjulf Owren. "Lie group methods for rigid body dynamics and time integration on manifolds". In: *Computer Methods in Applied Mechanics and Engineering* 192.3 (2003), pp. 421–438. ISSN: 0045-7825. DOI: https://doi.org/10.1016/S0045-7825(02)00520-0. URL: https://www.sciencedirect.com/science/article/pii/S0045782502005200.

[4] Elena Celledoni et al. *Lie Group integrators for mechanical systems*. 2021. arXiv: 2102.12778 [math.NA].

[5] The SciPy community. *NumPy API Reference. Machine limits for floating point types*. Jan. 2021. URL: https://numpy.org/doc/stable/reference/generated/numpy.finfo.html.

[6] Lawrence Conlon. *Differentiable Manifolds*. Birkhäuser Boston, 1993. DOI: 10.1007/978-1-4757-2284-0. URL: https://doi.org/10.1007/978-1-4757-2284-0.

[7] Charles Curry and Brynjulf Owren. "Variable step size commutator free Lie group integrators". In: *Numerical Algorithms* 82.4 (Jan. 2019), pp. 1359–1376. DOI: 10.1007/s11075-019-00659-0. URL: https://doi.org/10.1007/s11075-019-00659-0.

[8] J.R. Dormand and P.J. Prince. "A family of embedded Runge-Kutta formulae". In: *Journal of Computational and Applied Mathematics* 6.1 (1980), pp. 19–26. ISSN: 0377-0427. DOI: https://doi.org/10.1016/0771-050X(80)90013-3. URL: https://www.sciencedirect.com/science/article/pii/0771050X80900133.

[9] K. Engø and A. Marthinsen. "A Note on the Numerical Solution of the Heavy Top Equations". In: *Multibody System Dynamics* 5.4 (May 2001), pp. 387–397. ISSN: 1573-272X. DOI: 10.1023/A:1011459217639.

[10] Kenth Engø, Arne Marthinsen, and Hans Z. Munthe-Kaas. "DiffMan: An object-oriented MATLAB toolbox for solving differential equations on manifolds". In: *Applied Numerical Mathematics* 39.3 (2001). Themes in Geometric Integration, pp. 323–347. ISSN: 0168-9274. DOI: https://doi.org/10.1016/S0168-9274(00)00042-8. URL: https://www.sciencedirect.com/science/article/pii/S0168927400000428.

[11] The Python Software Foundation. *Python 3.9.5 documentation. Classes*. June 2021. URL: https://docs.python.org/3/tutorial/classes.html.

[12] The Python Software Foundation. *Python 3.9.5 documentation. Modules*. June 2021. URL: https://docs.python.org/3/tutorial/modules.html.

[13] The Python Software Foundation. *The Python Standard Library*. `unittest` – *Unit testing framework*. June 2021. URL: https://docs.python.org/3/library/unittest.html.

[14] E. Hairer. "Geometric Integration of Ordinary Differential Equations on Manifolds". In: *BIT Numerical Mathematics* 41.5 (Dec. 2001), pp. 996–1007. ISSN: 1572-9125. DOI: 10.1023/A:1021989212020. URL: https://doi.org/10.1023/A:1021989212020.

[15] Ernst Hairer. *Solving Differential Equations on Manifolds*. Lecture notes from "Équations diffèntielles sur des sous-variétés". Univserité de Genève, Section de mathématiques, 2-4 rue du Lièvre, CP64, CH-1211 Genève 4, June 2011. URL: https://www.unige.ch/~hairer/poly-sde-mani.pdf.

[16] Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I. Nonstiff Problems*. Springer Series in Computational Mathematics. Berlin, Heidelberg: Springer-Verlag, 1993. ISBN: 978-3-642-08158-3. DOI: 10.1007/978-3-540-78862-1.

[17] Ernst Hairer and Gerhard Wanner. "Runge–Kutta Methods, Explicit, Implicit". In: *Encyclopedia of Applied and Computational Mathematics*. Ed. by Björn Engquist. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 1282–1285. ISBN: 978-3-540-70529-1. DOI: 10.1007/978-3-540-70529-1_144. URL: https://doi.org/10.1007/978-3-540-70529-1_144.

[18] Ernst Hairer, Gerhard Wanner, and Christian Lubich. "Non-Canonical Hamiltonian Systems". In: *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 237–302. ISBN: 978-3-540-30666-5. DOI: 10.1007/3-540-30666-8_7. URL: https://doi.org/10.1007/3-540-30666-8_7.

[19] Brian C. Hall. *Lie Groups, Lie Algebras, and Representations. An Elementary Introduction*. Graduate Texts in Mathematics. Springer International Publishing, 2015. ISBN: 978-3-319-37433-8. DOI: 10.1007/978-3-319-13467-3. URL: https://doi.org/10.1007/978-3-319-13467-3.

[20] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[21] Darryl D Holm. "Geometric Mechanics. Part II: Rotating, Translating and Rolling". In: IMPERIAL COLLEGE PRESS, Apr. 2008. Chap. 8. DOI: 10.1142/p549. URL: https://doi.org/10.1142/p549.

[22] Arieh Iserles et al. "Lie-group methods". In: *Acta Numerica* 9 (2000), pp. 215–365. DOI: 10.1017/S0962492900002154.

[23] Erik André Jakobsen. *PyLie. A Python package for solving ordinary differential equations evolving on non-linear manifolds*. GitHub repository. 2021. URL: https://github.com/jakobsen/pylie/.

[24] Erik André Jakobsen. *PyLie. A Python package for solving ordinary differential equations evolving on non-linear manifolds*. Python Package Index project. 2021. URL: https://pypi.org/project/pylie/.

[25] Serge Lang. *Introduction to Differential Manifolds*. 2nd ed. New York, NY: Springer-Verlag, 2002. DOI: 10.1007/b97450. URL: https://doi.org/10.1007/b97450.

[26] Taeyoung Lee, Melvin Leok, and N. Harris McClamroch. *Global Formulations of Lagrangian and Hamiltonian Dynamics on Manifolds*. Springer International Publishing, 2018. DOI: 10.1007/978-3-319-56953-6. URL: https://doi.org/10.1007/978-3-319-56953-6.

[27] Awad Al-Mohy and Nicholas Higham. "A New Scaling and Squaring Algorithm for the Matrix Exponential". In: *SIAM Journal on Matrix Analysis and Applications* 31 (Jan. 2009). DOI: 10.1137/09074721X.

[28] Hans Munthe-Kaas. "High order Runge-Kutta methods on manifolds". In: *Applied Numerical Mathematics* 29.1 (1999). Proceedings of the NSF/CBMS Regional Conference on Numerical Analysis of Hamiltonian Differential Equations, pp. 115–127. ISSN: 0168-9274. DOI: https://doi.org/10.1016/S0168-9274(98)00030-0. URL: https://www.sciencedirect.com/science/article/pii/S0168927498000300.

[29] Hans Munthe-Kaas. "Lie-Butcher theory for Runge-Kutta methods". In: *BIT Numerical Mathematics* 35.4 (Dec. 1995), pp. 572–587. ISSN: 1572-9125. DOI: 10.1007/BF01739828. URL: https://doi.org/10.1007/BF01739828.

[30] Hans Munthe-Kaas. "Runge-Kutta methods on Lie groups". In: *BIT Numerical Mathematics* 38.1 (Mar. 1998), pp. 92–111. ISSN: 1572-9125. DOI: 10.1007/BF02510919. URL: https://doi.org/10.1007/BF02510919.

[31] Hans Munthe-Kaas and Antonella Zanna. "Numerical Integration of Differential Equations on Homogeneous Manifolds". In: *Foundations of Computational Mathematics*. Ed. by Felipe Cucker and Michael Shub. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 305–315. ISBN: 978-3-642-60539-0.

[32] Peter J. Olver. *Applications of Lie Groups to Differential Equations*. Springer US, 1986. DOI: 10.1007/978-1-4684-0274-2. URL: https://doi.org/10.1007/978-1-4684-0274-2.

[33] Brynjulf Owren. "Analytic functions on the ad-operator in $\mathfrak{se}(3)$". Unpublished note. Nov. 2020.

[34] Brynjulf Owren. *Lie group integrators*. Slides from a lecture given to the THREAD European Training Network. Norwegian University of Science and Technology, Trondheim, Feb. 2021. URL: https://wiki.math.ntnu.no/_media/thread/s-manifolds-final.pdf.

[35] William H. Press et al. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd ed. USA: Cambridge University Press, 2007. ISBN: 0521880688. URL: http://www.numerical.recipes/.

[36] Springer Verlag GmbH, European Mathematical Society. *Cayley-Hamilton theorem. Encyclopedia of Mathematics*. URL: https://encyclopediaofmath.org/wiki/Cayley-Hamilton_theorem.

[37] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. 1st ed. New York: Springer-Verlag. ISBN: 978-0-387-90420-7. DOI: 10.1007/978-1-4757-5592-3.

[38] Loring W. Tu. *An Introduction to Manifolds*. New York, NY: Springer New York, 2008, pp. 47–55. ISBN: 978-0-387-48101-2. DOI: 10.1007/978-0-387-48101-2_5. URL: https://doi.org/10.1007/978-0-387-48101-2_5.

[39] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[40] Eric W. Weisstein. *Bernoulli Number*. From MathWorld—A Wolfram Web Resource. URL: https://mathworld.wolfram.com/BernoulliNumber.html.

[41] Eric W. Weisstein. *Chaos*. From MathWorld—A Wolfram Web Resource. URL: https://mathworld.wolfram.com/Chaos.html.

[42] Menelaos Zikidis. *Homogeneous Spaces*. Lecture notes from Heidelberg University. 2016. URL: https://www.mathi.uni-heidelberg.de/~lee/MenelaosSS16.pdf.

Appendix A

Solving an equation on S^2 with PyLie

We will find a numerical approximation to the solution of the ODE

$$\dot{y}(t) = \begin{bmatrix} 0 & 0.5y_1 & -\sin t \\ -0.5y_1 & 0 & \cos t \\ \sin t & -\cos t & 0 \end{bmatrix} \begin{bmatrix} y_1(t) \\ y_2(t) \\ y_3(t) \end{bmatrix} := f(t, y) \cdot y, \tag{A.1}$$

$$y(0) = \begin{bmatrix} \sin 1.1 \\ 0 \\ \cos 1.1 \end{bmatrix} := y_0 \tag{A.2}$$

using PyLie. As f is skew-symmetric and $\|y_0\|_2 = 1$, $y(t) \in S^2$ for all t; see (2.1). It is assumed that PyLie is already installed and is available to the Python interpreter.

To begin, we import PyLie. We will also use NumPy [20] to implement the ODE.

```
import pylie
import numpy as np
```

Next, we implement $f(t, y)$. Recall that the matrix-vector product on the right hand side of (A.1) is the Lie group action of SO(3) on S^2, and should therefore not be included in the formulation of the problem to be used by PyLie.

```
def f(t, y):
    return np.array([
      [0,            0.5 * y[0], -np.sin(t)],
      [-0.5 * y[0],  0,           np.cos(t)],
      [np.sin(t),   -np.cos(t),   0]
    ])
```

Next, we define the necessary parameters for PyLie to solve the problem.

```
y0 = np.array([np.sin(1.1), 0, np.cos(1.1)])
t_start = 0
t_end = 5
h = 0.1
manifold = "hmnsphere"
method = "RKMK4"
```

With this in place, the equation is solved with a single line of code. The solution is accessed as an attribute on the returned object.

```
solution = pylie.solve(f, y0, t_start, t_end, h, manifold, method)
y = solution.Y
t = solution.T
```

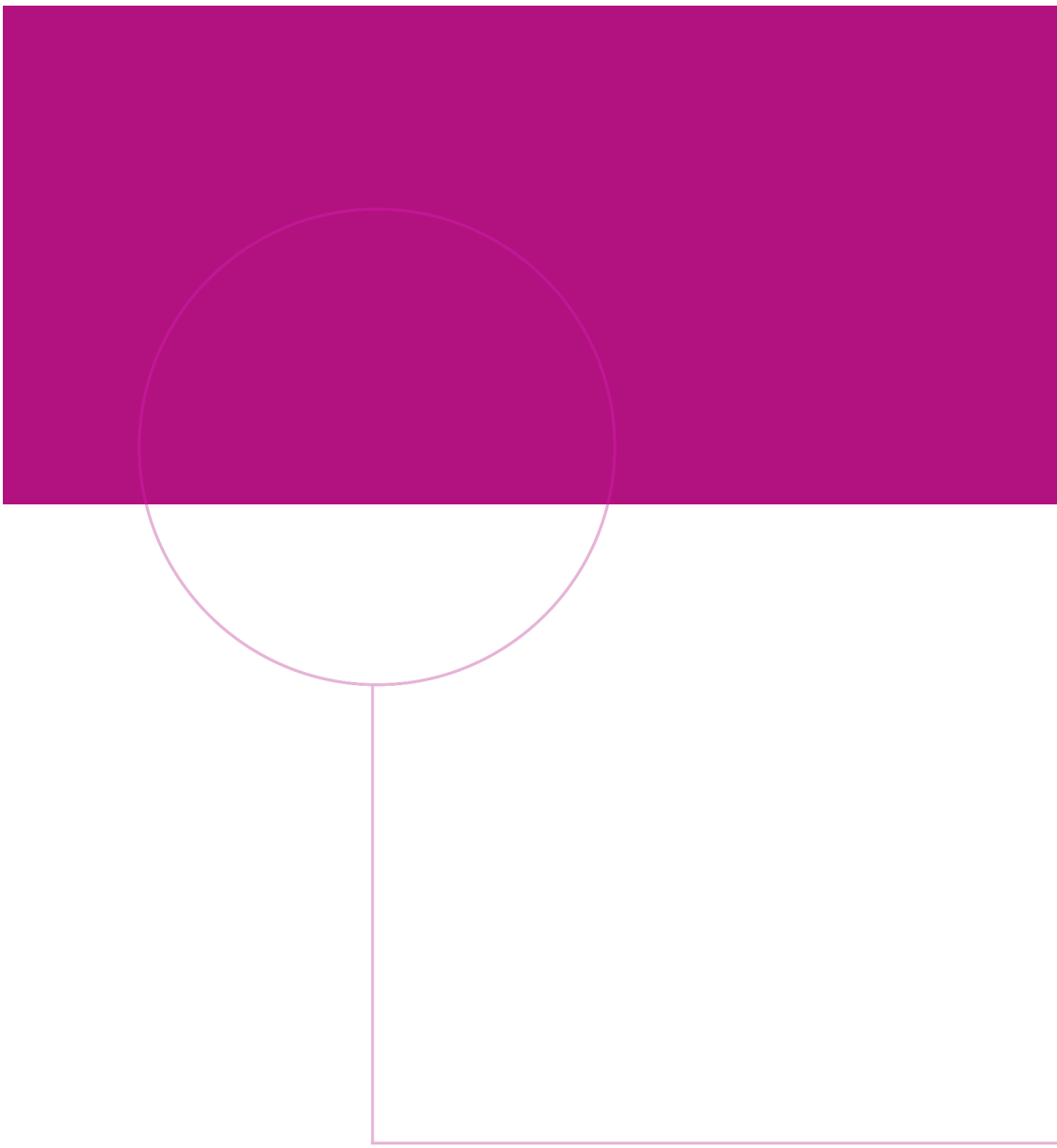
The complete code is shown in Listing 9. The conditional in line 15 ensures the code will only run if the file is called directly, as opposed to being imported into another Python file.

```python
import pylie
import numpy as np


def f(t, y):
    return np.array(
        [
            [0, 0.5 * y[0], -np.sin(t)],
            [-0.5 * y[0], 0, np.cos(t)],
            [np.sin(t), -np.cos(t), 0],
        ]
    )


if __name__ == "__main__":
    y0 = np.array([np.sin(1.1), 0, np.cos(1.1)])
    t_start = 0
    t_end = 5
    h = 0.1
    manifold = "hmnsphere"
    method = "RKMK4"
    solution = pylie.solve(
        f, y0, t_start, t_end, h, manifold, method
    )
    y = solution.Y
    t = solution.T
```

Listing 9: The complete code to solve (A.1) with initial value (A.2) using PyLie.



NTNU

Norwegian University of
Science and Technology